Clojure programming – the first two chapters

K. S. Ooi
Foundation in Science
INTI International University
Persiaran Perdana BBN, Putra Nilai
71800 Nilai, Negeri Sembilan, Malaysia
E-Mail: kuansan.ooi@newinti.edu.my

Abstract

I use Clojure to write scripts to explore and gray-hack the systems when I was asked to do so. Sadly, Clojure is not used widely by my fellow security experts; many of them, if they are required to write scripts, use Python instead. Clojure is simply, beautiful. However, Clojure has been recently relegated; it is no longer in top 25 popular programming languages at pypl (see http://pypl.github.io/PYPL.html) and has been listed as #51 to #100 programming language by TIOBE (https://www.tiobe.com/tiobe-index/). This article is the first two chapters of my note on Clojure programming. If you are new to this language, this can be your first introduction to Clojure, and hopefully you embrace it as your scripting language for your security work.

Keywords: Clojure, programming, scripting

1. Preface

This book is the first introduction to Clojure. I assume you have not learned programming before, let alone Clojure. This book teaches you programming, in general, and Clojure, in particular. If you had learned programming before, that obviously puts you in good position to understand this book, but that programming experience is not necessary. I will teach you, as I have claimed, programming in Clojure from ground up. However, you have to work. The subtitle itself suggests what you should ready yourself to type and run each code, and see its outcome. Do not stop there! I encourage you to change the code, and see how the outcome becomes different. By experimenting with the codes, you will learn, regardless of prior programming experience.

Let me reiterate. Learning to program requires participation. If you only read this book to learn Clojure, you are learning it from afar, as if you were watching a game as a spectator. The likelihood of success to get Clojure into your system this way is slim. You have to get involved.

This also means that you have to have a fair amount of skepticism. Do not believe all the things I say, try the codes. Please also try all the problems that sprinkle over the text. You should not miss any of them.

2. Installing Clojure

Clojure relies on Java; Clojure feeds on Java. Please google to find out how to get Clojure into your system.

3. Greeting from Clojure

Let us write the first line of code using Clojure, instructing it to print a message "From Clojure with love." Type the following code into the REPL, and press enter.

```
=>(println "From Clojure with love.")
;=>From Clojure with love.
;=>nil
```

The output is the message. After the message, another output is *nil*. This is actually good news. We will talk more about *nil* later.

Conventions used in input and output

When you type and try out the code in REPL

- type the code after **=>** and press enter
- the output prompt is denoted by `;=>`

Using the above example, you type `(println "From Clojure with love.")` after the prompt `=>`

```
=>(println "From Clojure with love.")
```

and press enter. The outputs that follow are

```
;=>From Clojure with love.
;=>nil
```

The prompt and output prompt used by REPL is different. Using REPL though lein or Nightcode, you may see

```
user=>(println "From Clojure with love.")
From Clojure with love.
nil
```

The `user` before `=>` is the current **namespace** of your Clojure code. When you run the code, do not jump from your seat if you see a different namespace. You might have run the code under different namespace.

Please take note about the difference.

Note that a pair of parentheses encloses a Clojure expression. You may call it *s-expression*, which is the short for symbolic expression. It is more fitting if we call it ***list***. Let me repeat again: in Clojure, a *list* is a sequence of objects within a pair of parentheses.

> **Run the code or evaluate the list?**
> When I say *run the code*, I actually mean **evaluate the list**. I will prefer the latter. Furthermore, when I say *try the code*, I actually mean *try evaluating the list*. Our purpose is to learn programming; so, let us move on.

In our first example, the list consists of *two* objects. The first one is the function *println*, a print line command. The second object is a *string* "From Clojure with love." The second object is the parameter to the first object, and these objects together form a list; the evaluation of this list results in printing the message. In this example, the *println* is the first object of the list. You will find that in many examples that follow, functions and operators are the first object of a list.

> **What is a function?**
> When you put a function as the first object of a list, you are directing Clojure to evaluate the list by *calling that function*. Each function in Clojure performs a different *task*. A function may or may not take parameters.
>
> **So, what is a string?**
> A string is an array of characters enclosed by a pair of double quotes ".

In our second example below, *println* takes more than one parameter of different types: strings "I have", "in my pocket.", and integer 7.

```
=>(println "I have" 7 "coins in my pocket.")
;=>I have 7 coins in my pocket.
;=>nil
```

Problem

Identify the function use in the following list.

```
=> (map - [5 6 3 0] [6 7 8 9])
;=> (-1 -1 -5 -9)
```

Solution

map

(You may or may not know what map does at this moment – this is not the avenue to discuss this function. Your job in this problem is to identify function used in the list.)

Problem

Identify the function used in the list below.

```
=> (dec 888)
;=> 887
```

Solution

dec

4. To evaluate or not to evaluate

There may be occasions that you *do not* want Clojure to evaluate a list; instead, you want Clojure to take the list *literally*. On those occasions, the right function to use is **quote**. For example, if you let Clojure run the following code, Clojure will print the sum of the two numbers.

```
=> (+ 3 4)
;=> 7
```

However, the following evaluation results in Clojure *outputting the list literally*, without evaluating it:

```
=> (quote (+ 3 4))
;=> (+ 3 4)
```

Clojure programmers commonly use the shorthand notation of quote, which is the apostrophe. An *idiomatic* Clojure programmer will likely rewrite the above list as follows:

```
=> '(+ 3 4)
;=> (+ 3 4)
```

Quoting a list has many uses. One of them is to delay an evaluation. In the following example, I first store the list in a variable `x` as quote, using the function **def**.

```
=> (def x '(+ 3 4))
;=> #'User/x
```

(I will talk more about def later.) Next, I can evaluate `x` using the function **eval**:

```
=> (eval x)
;=> 7
```

Let us go back to our `println` example. Note that `println` function inserts a **whitespace** (a blank) in between its parameters. Clojure provides us with a concatenation function **`str`** that takes in parameters of different types and returns a string without inserting whitespaces between them. For example

```
=>(str "Make" '- "my" 3 "day" '$)
;=>"Make-my3day$"
```

Note that when I put `'` to prefix the minus sign `-` , I am telling Clojure not to do any evaluation, that is do not perform subtraction in this case. In effect, I am telling Clojure to treat minus sign as a symbol. Try quote `3` with `'`, the code will work as well.

```
=>(str "Make" '- "my" '3 "day" '$)
;=>"Make-my3day$"
```

Try taking away `'` from the minus sign, Clojure will try to do subtraction, but to no avail, and hence, an error occurs:

```
=>(str "Make" - "my" '3 "day" '$)
;=>"Makeclojure.core$_@e091f47my3day$" ERROR ☹💣☠
```

Problem

What is the output from the evaluation of the following list?

```
=>(println (str "I have" 7 "coins in my pocket."))
```

Solution

```
;=>I have7coins in my pocket.
;=>nil
```

Problem

Show that `"I have"` and `"coins in my pocket."` are strings, whereas 7 is not. A string consists of a series of character, enclosed by a pair of `"`.

Solution

```
=>(string? "I have")
;=>true
=>(string? "coins in my pocket.")
;=>true
=>(string? 7)
;=>false
```

Problem

Show that the list `(str "I have" 7 "coins in my pocket.")` returns a string.

Solution

```
=>(string? (str "I have" 7 "coins in my pocket."))
;=>true
```

Problem

How do you find out that `'-` is a symbol, whereas `-` is not?

Solution

```
=>(symbol? '-)
;=>true
=>(symbol? -)
;=>false
```

Problem

When you prefix `'` to an object, you instruct Clojure not to do *evaluation*. Since Clojure does not evaluate numbers by themselves (for example, Clojure will evaluate expressions with numbers), `'3` and `3` are numbers, with or without `'`. Show this fact using Clojure.

Solution

```
=>(number? '3)
;=>true
=>(number? 3)
;=>true
```

Problem

We typically classify numbers into **floating point** and **integer**s. Floating-point numbers are numbers with decimals, for example 3.145 and 4.0. Integers are whole numbers, such as 30, -6 and 4500. When Clojure evaluate the following lists, specify whether it will evaluate each of them as **true** or **false**.

```
=>(number? -3.0)
=>(integer? -3.0)
=>(number? 9)
=>(integer? 9)
```

Solution

```
;=>true
;=>false
;=>true
;=>true
```

5. What a function does? How do I look up a function?

Each function in Clojure perform a specific *task*. If you had gone through examples and problems so far, you have had quite an experience already using functions. To use a function, you put the function name as the first object of a list, supply the function with the necessary arguments, and evaluate the list. But the question we always ask is how do we know what a function does.

Use the **doc** function.

Now, how do you learn more about the function **string?**? You *doc* the string?.

```
=> (doc string?)
;=> -------------------------
clojure.core/string?
([x])
  Return true if x is a String
nil
```

You may not understand the entire printout at this moment. I assure you that when you had covered about half of this book, you will. Reading the printout, you should at least know **doc** *prints documentation* for, in this case, the behavior of *string?*.

You can even able to find out what **doc** function itself does: you *doc* the doc. You will get printout for doc itself!

```
=> (doc doc)
;=> -------------------------
clojure.repl/doc
([name])
Macro
  Prints documentation for a var or special form given its name
nil
```

To find out more about functions of Clojure's core library, pay a visit to

https://clojuredocs.org/clojure.core

or

https://clojuredocs.org/quickref

Problem

Find out what the function **dec** does using doc.

Solution

Using

```
=> (doc dec)
```

Read the printout, which is not shown here. Obviously, *dec* is short for **decrement**. The function decrements its argument, which means the function returns the argument one less. After the decrement, 888 becomes 887. So, the output of `(dec 888)` is 887, and that of `(dec -90)` is -91.

Problem

Find out what the function **inc** does.

Solution

Using

```
=> (doc inc)
```

The function **inc** is the opposite of dec. Therefore, inc will **increment** a number. The output of the list `(inc 99)` is 100, and that of `(inc -99)` is -98.

6. Arithmetic operations

The following table summarizes five common arithmetic operators.

Symbol	Meaning	Example (;=>output)	Explanation
+	Addition	`(+ 4 9) ;=> 13`	4 + 9 = 13
–	Subtraction	`(- 8 15) ;=> -7`	8 – 15 = -7
*	Multiplication	`(* 12 6) ;=> 72`	12 * 6 = 72
/	Division	`(/ 6 3.0) ;=> 2.0`	6 / 3.0 = 2.0
rem	Remainder/Modulus	`(rem 24 7) ;=> 3`	24 % 7 = 3

How to test the example?

When you test the examples, please do not type the output. For example, when you test the example for subtraction,

```
(- 8 15) ;=> -7
```

you type

```
=>(- 8 15)
```

and press ENTER. You will see the output. Please test all the examples.

What is a prefix notation?

The arithmetic expressions you learn from school are in **infix** notation. The expressions in the explanation column of the above table are also in this notation. Let me use an example. If you want to add two numbers, you put the first number on the left, the addition operator in the middle, and the second number on the right.

6 + 4

Clojure uses the **prefix** notation to express arithmetic expression. When I express the above addition using this notation in Clojure, it becomes

(+ 6 4)

There is another notation, the postfix, but its coverage is beyond the scope of this book.

Two of the arithmetic operations call for further explanation. First the division. When you write

```
=> (/ 3 5)
;=> 3/5
```

However, if one or both of the numbers are real, the output will also be real. For example:

```
=> (/ 3.0 5)
;=>0.6
=> (/ 3 5.0)
;=>0.6
=> (/ 3.0 5.0)
;=>0.6
```

Why Clojure behaves this way? We must first find out how Clojure sees numbers under its hood. You will find that integers are Long, floating-point numbers are Double, and rational numbers are Ratio. I found these using the function `type`.

```
=> (type 3)
;=>java.lang.Long
=> (type 4.5)
;=>java.lang.Double
=> (type 3/5)
;=>clojure.lang.Ratio
```

Long is 64-bit integer. Double is double-precision 64-bit IEEE 754 floating point. **Ratio** is interesting. When you perform division, Clojure will do its level best to avoid precision lost. Clojure will rather keep 1/3 than, say, 0.3333 (precise to 4 significant figures), at the expense of performance. Let us try some lists.

From your high school math, when you divide 24 by 21, you can express the answer in rational number (a ratio):

$$\frac{24}{21} = \frac{8\times3}{7\times3} = \frac{8}{7}$$

Clojure does this for you if you use whole numbers:

```
=> (/ 24 21)
;=> 8/7
```

You can also make a ratio out of a real number:

```
=> (rationalize 0.35)
;=> 7/20
```

Using your high school math:

$$0.35 = \frac{35}{100} = \frac{7\times5}{20\times5} = \frac{7}{20}$$

Computation involving **modulus** or **remainder** is another interesting arithmetic operation I want to explore here. In high school, you find the remainder of the division of 45 by 6 using the following method:

```
      7
   ------
6 | 45
    42
   ---
     3
```

The remainder is 3

Clojure computes the remainder of 45 by 6 using the **rem** function:

```
=> (rem 45 6)
;=> 3
```

Remainder of a division has many uses. You can determine whether a number is divisible by another number. Because 46 is not divisible by 3, the remainder is not *equal to* zero. Therefore,

```
=> (= (rem 46 3) 0)
;=> false
```

To check whether a number (say 8) is even, that is the number is divisible by 2, you may try

```
=> (= (rem 8 2) 0)
;=> true
```

> Clojure provides a built-in function **even?** to do this
>
> ```
> => (even? 8)
> ;=> true
> ```

Problem

The following evaluation results in a ratio.

```
=> (* 5 (/ 2 3))
;=> 10/7
```

How do you coerce Clojure to output the result in real number?

Solution

I just make one of the numbers a real number; in the following case, I use 5.0 instead of 5.

```
=> (* 5.0 (/ 2 3))
;=> 3.333333333333334
```

You can convert any number in the list (5, 3, or 2) to a real number to get the desired result; you can also convert any two of them to real numbers, or all three of them.

Problem

How many quarters can one get from $5.30? (FYI: a quarter worth 25 cents – that is a quarter of $1.)

Solution

If you use division, you will get the wrong answer.

```
=> (/ 5.30 0.25)
;=> 21.2
```

You need the **quot**ient of the division. In Clojure, you should use the **quot** function.

```
=> (quot 5.30 0.25)
;=> 21.0
```

You will get 21 quarters. If you want a whole number as your answer, you can either

```
=> (int (quot 5.30 0.25))
;=> 21
```

or

```
=> (quot 530 25)
;=> 21
```

The **int** function *coerce*s or converts numbers to integers.

7. Arithmetic expressions – using Clojure as a calculator

So far, we have been dealing with simple arithmetic expressions. Many expressions you are going to deal with, however, involve more than one operator.

Operator precedence and associativity

Let us take a step back, and use the *infix notation* to compute the following expression, which has three arithmetic operators (-, /, and *):

$$2 - 6/12 * 3$$

There is a "universal" rule to carry out the computation. Ask your engineering friends, they might know this rule. The rule is governed by operator precedence and associativity. The following table only lists two groups of operators in decreasing precedence levels.

Group	**Operator**	**Associativity**
Multiplicative	* / rem	Left to right
Additive	+ -	Left to right

According to the rule, / and * have *higher precedence* than -, which means one would compute / and * first, that is

$$6/12 * 3$$

before the subtraction. However, /and * have the same level of precedence, and therefore you must invoke their associativity to help decide which of these two operators will be carried out first. According to the rule, operators / and * *associate* from left to right. Hence, you compute 6/12*3 from left to right:

$$6/12 * 3 \;\; = 0.5 * 3 = 1.5$$

In the end you compute the subtraction, and the answer is

$$2 - 1.5 = \mathbf{0.5}$$

If you compute the multiplication first, and thus *defying the rule*, the answer would be different:

$$2 - 6/(12 * 3) \ = 2 - 6/36 \ = \mathbf{1.833333333333334}$$

Or, if you care less about the precedence, and you take all operators associate from left to right: you carry out subtraction, followed by division and finally multiplication. The answer will be -1!

Prefix notation used by Clojure *does not have to face the issue of operator precedence and associativity*. Let say you want to compute the expression 2 – 6/12*3 and you want to follow the rule. First, you perform the division: you type

```
=> (/ 6.0 12)
```

Next, you perform the multiplication: so you expand the preceding expression to

```
=> (* (/ 6.0 12) 3)
```

Finally, you expand again the expression to include the subtraction:

```
=> (- 2 (* (/ 6.0 12) 3))
```

In summary, the Clojure list to compute the expression 2 – 6/12 * 3 is

```
=> (- 2 (* (/ 6.0 12) 3))
;=> 0.5
```

Notice that the innermost parenthesis is the first operation you carry out.

Problem

What is the Clojure list to compute 4*5 – 6 – 5?

Solution

```
=> (- (- (* 4 5) 6) 5)
;=> 9
```

Which operation will be first carried out? The multiplication. Hence, you type

```
=> (* 4 5)
```

Since the subtraction is associated from left-to-right, you expand the list as follows:

```
=> (- (* 4 5) 6)
```

Finally,

```
=> (- (- (* 4 5) 6) 5)
```

Problem

The formula to convert temperature in Fahrenheit (F) to Celsius (C) is:

$$C = \frac{5}{9} \times (F - 32)$$

A student in California recently reported that the hottest temperature of the year was 73.7 degrees Fahrenheit. What is this temperature in Celsius? Use Clojure to compute.

Solution

I compute (F – 32) first, where F = 73.7, followed by multiplication by 5, and finally, division by 9. Therefore,

```
=> (/ (* 5 (- 73.7 32)) 9)
;=> 23.166666666666668
```

Problem

A liquid mixture containing sulfur boils at 118 degrees Celsius. What is the boiling temperature in Fahrenheit? Use Clojure to compute. The conversion is given by:

$$F = \frac{9}{5} \times C + 32$$

Solution

Since 9/5 nicely yields 1.8, without loss in precision, we write the formula as

$$F = 1.8 \times C + 32$$

This will help us skip one operation. So, instead of

```
=> (+ (/ (* 9 118) 5) 32.0)
;=> 244.4
```

we write

```
=> (+ (* 1.8 118) 32)
;=> 244.4
```

8. Looking ahead – define function using defn

In the last section, we have used Clojure as a calculator to perform temperature conversions. That is fine if you do that calculation once in a blue moon. Had you work for a meteorological office, a place at which you do excessively many temperature conversions, you need something better. I mean it is high time to come up with a better way to do the conversion using Clojure. If you cannot find an already built-in function, one of the way is to define the function using defn.

For the conversion of Fahrenheit to Celsius, we could have defined a function called F-2-C, with a **parameter** *temp-F* that stands for temperature in Fahrenheit:

```
=> (defn F-2-C
     [temp-F]
     (/ (* 5 (- temp-F 32)) 9.0))
```

Within that namespace, we can do the temperature conversion multiple times, by using the following Clojure lists. The first object is F-2-C, the function, and the second object is the temperature in Fahrenheit as the **argument** to the function:

```
=> (F-2-C 73.7)
;=> 23.166666666666668
=> (F-2-C 128)
;=> 53.333333333333336
=> (F-2-C 37)
;=> 2.7777777777777777
…
```

We will cover function in more detail later in chapter 2. Do not worry if you do not understand function definition fully yet. You will encounter defn very often as a Clojure programmer. You may wonder why I give you a short preview of function. It is important that you start thinking about writing organized, modular code. By giving you this preview, I am trying to make you aware that programmers do not use Clojure as calculator per se; Clojure can do much more!

In the meantime, try the following problem.

Problem

Define a function called **C-2-F**, with a parameter *temp-C* that stands for temperature in Celsius. Test the function using a number of temperatures.

Solution

```
=> (defn C-2-F
       [temp-C]
       (+ (* 1.8 temp-C) 32))

=> (C-2-F 118)
;=> 244.4
=> (C-2-F 0)
;=> 32.0
…
```

9. Java interop

I leave this one out at this moment.

10. Compound interest

When your bank pays interest on the original amount of money you deposited as well as interest your account has already earned, the sum of money you have, *A*, after *t* years at interest rate *r* is given by

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

This is the famous compound interest equation, where *n* is the number of times per year your bank compounds the interest.

Let us pluck in some numbers to the equation. If your bank compounds the interest twice a year (*n* = 2) and pays the interest at the rate of 2% (*r* = 0.02), after three years (t = 3) for the original amount $2000 (principal *A* = 2000) you deposited, you will have in your account

$$A = 2000 \times \left(1 + \frac{0.02}{2}\right)^{2\times 3} = 2123$$ (I round the number to 4 significant numbers)

The equation requires us to compute the power of a number. Let us call Java.lang.Math.pow method to do so. To compute 2^5, we write

```
=>(. Math pow 2 5)
;=> 32.0
```

Using this knowledge, In Clojure we compute the amount of money in the bank by

```
=>(* 2000 (. Math pow (+ 1 (/ 0.02 2.0)) (* 2 3)))
```

Using this knowledge

11. An array of things – Collections

I leave this one out at this moment.

12. Simple function definition using fn, def and defn

In chapter 1, if you carry out the experimentation of all examples and problems, you have clocked in an impressive hours of using functions, but not defining functions. You did have an opportunity in defining two functions for temperature conversions. You will accrue more experience of this skill in this chapter.

However, I will delay the discussion on first-class and higher-order function until later chapters. Instead, we start this section by defining a trivial function, a function that returns two times the value of its argument. In addition to that, this function has no name. It looks like this:

```
=> (fn [x] (* 2 x))
```

Note that

- We use **fn** to define the function.
- The function has no name; it is an **anonymous** function.
- The function has one **parameter**, `x`. The parameter is enclosed in a pair of square brackets [].
- The **body** of the function is `(* 2 x)`. As expected, it is a list. In this case, the list computes the two times value of `x`.

When we use the anonymous function, we must supply it with **argument(s)**. Let us compute the double of 8:

```
=> ((fn [x] (* 2 x)) 8)
;=> 16
```

The argument is set after the definition. Clojure provides a **reader macro** that reduces the number of parentheses. We will call the macro the shorthand notation. You can obtain the same result using the shorthand notation:

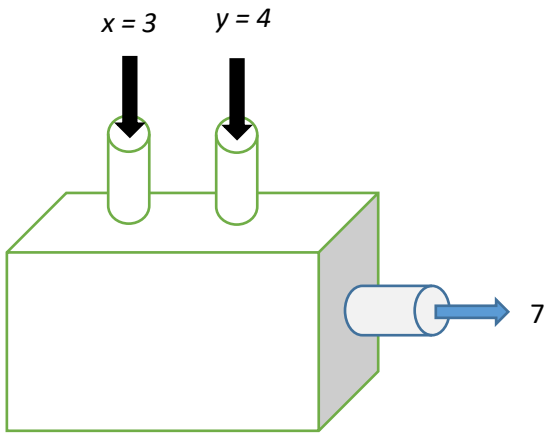
```
=> (#(* 2 %) 8)
;=> 16
```

Now the code looks less congested, with less one pair of parentheses. You may want to object the use of the shorthand notation. Unfortunately, many Clojure programmers have the habit of using it in regular basis. This shorthand notation has become idiomatic.

Let us work on another anonymous function, with two parameters. This is, again, a trivial function. It returns the sum of the two arguments. In this example, the two arguments are 3 and 4, respectively:

```
=> ((fn [x y] (+ x y)) 3 4)
;=> 7
```

You may view the function using the following figure. From the caller perspective, a function call is a black box to which you input the arguments, and out from the box the result. The user/caller of the function may use *doc* to find out how to use the function (in Chapter 1, you use a number of functions this way), but the implementer (in this chapter) must know how the black box works.



Using the shorthand notation, we rewrite the function as follows:

```
=> (#(+ %1 %2) 3 4)
;=> 7
```

When you have more than one parameter, you cannot use % alone. You have use number after %.

Problem

Define an anonymous function that returns the argument tripled. Test the function using the argument 45; the returned value should be 135.

Solution

It can be

```
=> ((fn [x] (* 3 x)) 45)
;=> 135
```

Or

```
=> (#(* 3 %) 45)
;=> 135
```

Problem

The following anonymous function returns the sum of squares of three arguments. When the arguments are 3, 7 and 5, the sum of squares is $3^2 + 7^2 + 5^2 = 83$.

```
=> ((fn [x y z] (+ (* x x) (* y y) (* z z))) 3 7 5)
;=> 83
```

Rewrite the function using the shorthand notation.

Solution

```
=> (#(+ (* %1 %1) (* %2 %2) (* %3 %3)) 3 7 5)
;=> 83
```

Problem

The solution of

$$ax^2 + bx + c = 0$$

is

$$x = \frac{-b \pm \sqrt{b^2 - 2ac}}{2a}.$$

Let say we are interested in one of the solution,

$$x = \frac{-b - \sqrt{b^2 - 2ac}}{2a}$$

when

$$a = 5, b = 7, c = 2.$$

We either have ❶

```
=> (#(/ (- (* -1 %2) (Math/sqrt (- (* %2 %2) (* 4 %1 %3) ))) (* 2 %1)) 5 7 2)
;=> -1.0
```

or we factor out the square root part as an anonymous function ❷:

```
=> (/ (- -7 (#(Math/sqrt (- (* %2 %2) (* 4 %1 %3))) 5 7 2)) (* 2 5))
;=> -1.0
```

Which do you think is better, ❶ or ❷?

Solution

If we use Clojure as a calculator, either one will do – I would have used a real calculator. The Clojure folks would be very, very sad if we stop learning Clojure at this point.

Let us take a second look at both codes. The advantage of using anonymous function in both codes is not that apparent. What does that mean? The main purpose of using anonymous function should be to make the existing code easier to read. I don't see we are reaping such benefit here, but you may prefer one to another. This is rather subjective. Let us move on.

Imagine you work for a company that makes *cubic* fish tanks. It is not hard to speculate that one of the questions customers would frequently ask is how heavy a tank would be with water full tank. Situation like this requires you to compute the cube of the tank many times from its length. Let us first define an anonymous function:

```
=> (fn [x] (* x x x))
```

When you use it to compute the cube with length 2 m, you do

```
 => ((fn [x] (* x x x)) 2)
;=> 8
```

or

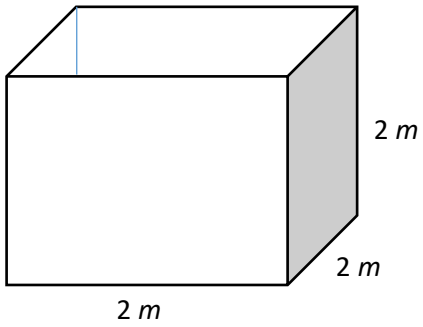
```
=> (#(* % % %) 2)
;=> 8
```

You estimate that the density of water at 20 °C is

998 kg/m^3.

Hence, the mass of water of the tank is

$$998\frac{kg}{m^3} \times 8m^3 = 7984\ kg.$$



A cubic fish tank has equal length, width and height. In this example, the tank is 2x2x2 m^3. The volume is 8 m^3. Density of water varies with temperature. To *estimate* the mass of water in the tank, we use pure water, which has a density of 998 kg/m^3 at 20 °C. If you are familiar with the factor-label method, the mass of the water in the tank can be estimated from:

$$998\frac{kg}{m^3} \times 8m^3 = 7984\ kg$$

Let us start with a rather sad solution: using Clojure as a mere calculator. We just multiple the two numbers:

```
=> (* 8 998)
;=> 7984
```

Or we do not separate the calculation of the volume of the cubic tank and the calculation of the mass of water. We define an anonymous function that calculates the volume. Next, we supply the function with an argument, and in turn use the returned value as the first argument for the multiplication with the second argument (density of water):

```
=> (* (#(* % % %) 2) 998)
;=> 7984
```

This is also sad. Why don't we assign the anonymous function a symbol called `mass-water-kg`, using **def**? Let us try that:

```
=> (def mass-water-kg
     (fn [length, density]
       (* length length length density)))
=> (mass-water-kg 2 998)
;=> 7984
```

The anonymous function is now bound to the symbol `mass-water-kg`. Within the namespace, you may call the function as many times as you like to compute the mass of water that can be contained in the cubic fish tank with different inputs in length and density. For example,

```
=> (mass-water-kg 5 998)
;=> 124750
=> (mass-water-kg 6 998.42)
;=> 215658.72
…
```

The user of this function will have many questions. Remember how we use **doc** to find out what a function really does. You may document the function this way:

```
=> (def mass-water-kg
    "([length, density])
    Returns the mass of water in kg from a cubic fish tank
    with 'length' in meter (m), and water 'density' in kg/m3."
     (fn [length, density]
       (* length length length density)))
```

So that when you *doc* the function within the namespace,

```
=> (doc mass-water-kg)
-------------------------
user/mass-water-kg
  ([length, density])
 Returns the mass of water in kg from a cubic fish tank
 with 'length' in meter (m), and water 'density' in kg/m3.
nil
```

Problem

Rewrite the `mass-water-kg` function using the shorthand notation (reader macro).

Solution

```
=> (def mass-water-kg
     "([length, density])
     Returns the mass of water in kg from a cubic fish tank
     with 'length' in meter (m), and water 'density' in kg/m3."
      #(* %1 %1 %1 %2))
```

Binding an anonymous function to a symbol – and hence to make it no longer anonymous – is one of the common tasks we do daily with Clojure, so much so that there available a macro, **defn**, that subsumes def and fn. Let us consider the *cube* function.

The cube function can be implemented in

```
=> (def cube (fn [x] (* x x x)))
```

or in shorthand notation:

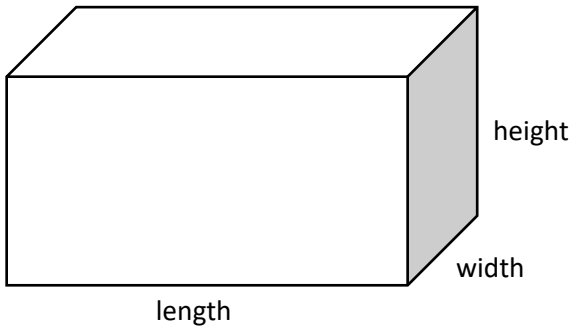
```
=> (def cube #(* % % %))
```

Now, using defn, we have

```
=> (defn cube [x] (* x x x))
```

Problem

Cubic fish tank, a special case of rectangle fish tank, has equal length, width and height. To generalize the solution, we shall work with rectangle fish tank.



In this problem, calculate the total mass of the fish tank from the input of length, width, height, density of fresh water at 20 °C, and the mass of the tank without water in kg.

Solution

First, I use defn to create a function that calculates the volume of the rectangular fish tank.

```
=> (defn volume-rectangle
      [length, width, height]
      (* length width height))
```

Second, I create a function that calculates the mass of water in the tank.

```
=> (defn mass-water
     [volume density-water]
     (* volume density-water))
```

Third, a function that calculate the total mass of the tank when it is full with water.

```
=> (defn total-mass
      [length width height density-water mass-fish-tank]
      (+ mass-fish-tank (mass-water (volume-rectangle length width height)
         density-water)))
```

To test,

```
=> (total-mass 4 2 1 998 700)
;=> 8684
```

13. Multi-arity function

You can define a function that behaves differently if you supply it with different number of arguments. We call this kind of function **multi-arity function**. This means that a multi-arity function has different versions, each of which defined for a specific number of argument.

In the following example, the first version of *greeting* is without any parameter, which we call one-arity (or 1-arity) version. The 2-arity version of *greeting* is for one parameter, and the 3-arity version is for two parameters. Notice that both the 1-arity and 2-arity versions call the 3-arity version, and hence the default version is the 3-arity's.

```
(defn greeting
  ([] (greeting "" ""))
  ([name] (greeting "" name))
  ([title, name] (println "Hi" title name "nice to meet you.")))
```

So, if you call *greeting* without an argument, you actually call the 3-arity version with both *title* and *name* set to blanks.

```
=> (greeting)
;=> Hi   nice to meet you.
;=> nil
```

However, if you call greeting with a single argument, you actually call the 3-arity version with *title* set to a blank.

```
=> (greeting "Phil")
;=> Hi  Phil nice to meet you.
;=> nil
```

The default version is the 3-arity's, which takes two parameters. Hence if you call *greeting* with two arguments:

```
=> (greeting "Dr." "Phil")
;=> Hi Dr. Phil nice to meet you.
;=> nil
```

Problem

A store is having an anniversary sale. All items are given 30% discount. The owner of the store can give an additional special discount if she wants to. Write a function called *anniversary-sale-price* to reflect the requirements.

The math you need:

Let *x* be additional discount, on top of 30%. The total discount is given by

$$price \times \left(\frac{30+x}{100}\right)$$

The discounted price is given by

$$price - price \times \left(\frac{30+x}{100}\right)$$
$$= price \times \left[1 - \left(\frac{30+x}{100}\right)\right]$$

Solution

```
(defn anniversary-sale-price
  ([price] (anniversary-sale-price price 0.0))
  ([price, more-discount] (* price (- 1.0 (/ (+ 30.0 more-discount) 100.0)))))
```

Let us call the function for an item with a price tag of $80. Without the additional discount, we have

```
=> (anniversary-sale-price 80)
;=> 56.0
```

With an additional discount of 5%, we have

```
=> (anniversary-sale-price 80 5)
;=> 52.0
```

14. Create a var using def

When you define an object in Clojure using **def**, you create a **var**. A var has to be associated with a **namespace**. For example, when you use def as follows

```
=> (def greeting "Hello")
;=> #'user/greeting
```

you create a var `user/greeting`. The **symbol** of the var is `greeting`. The symbol `greeting` in the namespace `user` is bound to the string `"Hello"`. Within the namespace `user` (yours may be different), if you type `greeting`, `"Hello"` will be returned.

```
=> greeting
;=> "Hello"
```

Now I may use `greeting` to greet as many people as I like:

```
=> (println greeting "Ms. Jane")
;=> Hello Ms. Jane
;=> nil

=> (println greeting "Sensei Tanaka")
;=> Hello Sensei Tanaka
;=> nil
…
```

What a lingo! We will simply say assign `"Hello"` to `greeting`. What we really mean is

- We use def to create a var
- The var has a symbol, which is `greeting`
- The symbol is bound to `"Hello"`
- Once defined, the var is available anywhere within the namespace.

We will discuss other characteristics of var when the need arises.

Problem

We return to compound interest in this problem. The accrued amount, also known as the future value, is given by

$$A = P \times \left(1 + \frac{r}{n}\right)^{nt}$$

A potential customer wanted to find out how much principal he or she should deposit to earn certain value in the future, your calculation will be

$$P = {A}\Big/{\left(1 + \frac{r}{n}\right)^{nt}}$$

The two calculations involve a common term called the *future value factor*, which we denote `future-factor`:

$$\texttt{future-factor} = \left(1 + \frac{r}{n}\right)^{nt}$$

Let say the bank compounds the interest quarterly, *n = 4*, and pays the interest of 1.8%, *r = 0.018*. For a period of 5 years, *t = 5*, find

(a) the future value A if one starts with the principal $5,000, and
(b) the principal P one must invest to earn a future value of $15,000.

Solution

```
=> (def future-factor (Math/pow (+ 1 (/ 0.018 4)) (* 4 5)))
;=> #'user/future-factor

=> (* 5000 future-factor)
;=> 5469.766990934119

=> (/ 15000 future-factor)
;=> 13711.735824269837
```

Problem

You may not like the solution of the preceding problem. First, you use Clojure as a mere calculator. Clojure has much wider usage. Second, `future-factor` is a constant. If the calculation is to be repeated with other *n*, *r* and *t* values, you have to define other constants. In this problem, define a `future-factor` **function**, with *n*, *r* and *t* as its parameters. Use the function to recalculate the values A and P of the preceding problem.

Solution

I define a function called `future-factor`.

```
(defn future-factor
  [n, r, t]
  (Math/pow (+ 1 (/ r n)) (* n t)))
```

I create a constant from the function whose arguments are *n = 4*, *r = 0.018*, and *t = 5*.

```
(def FUTURE-FACTOR1 (future-factor 4 0.018 5))
```

Finally, I use the constant to carry out the necessary calculations:

```
(* 5000 FUTURE-FACTOR1)
(/ 15000 FUTURE-FACTOR1)
```

I have tried to imagine writing a piece of software that solves a real-world problem but, along the way, that does not make any kind of decision.

It is indeed difficult to construct a practical computer program that solves problems but that does not make any kind of decision along the way. When your program is prompted to make a decision at a juncture, a particular expression will be evaluated to be either true or false. This kind of expression is called **Boolean** expression. The behavior of your program will change upon taking a course of action based on this Boolean expression: in the event that the evaluated value is true, your program will do something, which is different from what it will do otherwise.