Programming Assignment 2: Clojure

Beatrice Åkerblom
beatrice@dsv.su.se

November 27, 2025

1 Introduction

The assignment should be done individually.

For tutoring of all kinds of question regarding this assignment, please use the iLearn tutoring forum.

2 The Assignment

The assignment consists of two parts. The first part is an exercise in functional programming through defining a number of specified functions. The second part is an exercise in implementing Clojure macros. To get a grade higher than C, you need to solve both parts of the assignment.

2.1 Functional Programming

1. `my-max`

   "The `my-max` function takes a list of numbers and returns the largest number in the list"

   Please note that wrapping the built-in `max` function is not an acceptable way to implement `my-max`.

   ```
   > (my-max '(1 2 3 55 4 5 6))
   55

   > (my-max '(36))
   36

   > (my-max '())
   nil
   ```

2. `my-map`

   "The `my-map` function takes two arguments as input. The first element should be a list and the second element should be a function and the

return value is a list where the elements are the results of applying the input function on the elements of the input function.

Please note that wrapping the built-in `map` function is not an acceptable way to implement `my-map`.

```
> (my-map '(1 2 3 4 5 6) inc)
(2 3 4 5 6 7)

> (my-map '(-3 6 -9 12) pos?)
(false true false true)
```

3. `my-checksum1`

   "The `my-checksum1` function takes an argument that is a list of lists containing numbers. The output from the function will be a checksum value for the entire input."

   For each internal list, you need to calculate the difference between the largest value and the smallest value. The checksum for the entire input is the sum of all of these differences.

   Given the input

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

   The first internal list's largest and smallest values are 9 and 1, and the difference is 8.

   The second internal list's largest and smallest values are 0 and 0.

   The third internal list's largest and smallest values are 8 and 4, and their difference is 4.

   The fourth internal list's difference is 7.

   The fifth internal list's difference is 0.

   The sum of the differences is 19.

   ```
   > (my-checksum1 '((9 1 3 9 5) (0) (6 8 5 4) (1 3 4 6 10) (7 7 7 7)) )
   21
   ```

4. `my-checksum2`

   "The `my-checksum2` function takes three arguments. The first one is a list of lists containing numbers. The second is a function that calculates the checksum number of a single list. The third one is a function that operates on the checksums of the individual lists to produce the final checksum. The output from the function will be a checksum value for the entire input."

   For each internal list, you need to use the function from the second argument to calculate its value. For each output from the second argument function, you need to apply the argument function from the third argument.

   If the function in the second argument is the `my-max` function from above, and the function in the third argument is the is the `-` (that is the built-in Clojure subtraction function), running the function could look like this:

```
> (my-checksum2 '((9 1 3 9 5) (6 8 5 4) (7 7 7 7)) my-max -)
-6

> (my-checksum2 '((1 3 9 5) (6 5 4 3) (3 5 1)) my-max -)
-2
```

5. `my-reverse`

   "The `my-reverse` function takes a list of elements as input and returns a list with the same elements but in reversed order. If the input is not a list, `nil` is returned."

   ```
   > (my-reverse '(1 2 3 4 5 6))
   (6 5 4 3 2 1)

   > (my-reverse '(36))
   (36)

   > (my-reverse '())
   ()
   ```

   When implementing `my-reverse` you must use recursion, and you may not use the built-in reverse function.

2.2 Safe Macro

This part of the assignment is to write a macro called `safe` that is intended to function similarly to `try` in Java 7, and `using` in C#. It should be possible to use the macro on a single expression, or an expression with a binding form (i.e. `[variable value]`) that binds to an instance of a closable (see examples below).

Example, Java 7 `try`

```
try (Socket s = new Socket()) {
  s.accept();
} catch(Exception e) {}
```

In the code above the socket 's' is automatically closed, just as if you had explicitly written a finally clause

```
finally {
 if (s != null) s.close();
}
```

You should define a macro 'safe' that takes two arguments. The first argument should be a vector with two elements, one variable and one value (corresponding to 's' and 'new Socket()' in the example above). The second argument should be a form (expression) that should be evaluated.

If an exception is thrown inside the form the exception should be returned. Otherwise the evaluated value of the form should be returned. If a binding is provided it should be possible to use the bound variable inside the form. After execution of the form any variables that are bound shall be closed (e.g. `(let [s (Socket.)] (. s close))`) Note that closeable classes in Java implements the `Closeable`–interface (hint: use type hints). The return value of the macro is either the return value of the executed form or an exception.

The Clojure macro shall function in the following way:

```
user>  (import java.io.FileReader java.io.File)
java.io.File
...
user> (def v (safe (/ 1 0)))
user> v
#<ArithmeticException java.lang.ArithmeticException: Divide by zero>
user> (def v (safe (/ 10 2)))
user> v
5
user> (def v (safe [s (FileReader. (File. "file.txt"))] (.read s)))
user> v
105 ; first byte of file file.txt
user> (def v (safe [s (FileReader. (File. "missing-file"))] (. s read)))
user> v
#<FileNotFoundException java.io.FileNotFoundException:
missing-file (No such file or directory)>
```

2.3 Reflections

As a commented out text at the end of the file that should contain the safe macro, discuss how the macro you have implemented could have been implemented as a function. Is that even possible? Why?, Why not? If you have not implemented the safe macro, you could still discuss when you should and should not use macros and what is possible to solve with a function and macro respectively from a theoretical point of view. (max 300 words).

3 Handing in

All handed in files should contain the names of all who have participated in the work on a commented out line at the top of the file.

Part one should be handed in in a file called `functional.clj`.

Part two should be handed in in a file called `safe.clj`.

4 Grading

In order to receive the grade E you have to implement at least three of the specified functions to be OK.

To receive a grade better than C, you also have to implement the safe macro and write the reflection on macros.

Grading programming is more an art than a science. In the general case, it is extremely difficult to impossible to say that one program is better than the other. For the obvious reasons, it is impossible to cover the grading criteria completely. In any case, the points below are not totally black/white. This is why you should also should reflect on you implementation.

For the programming assignment for block 4 the following table will be used when grading the assignments.

Grade	Functions	Safe macro	Reflections
A	Good	Good	Good
B	Good	OK	OK
C	Good	Missing/OK-	OK-
D	OK	Missing/OK-	Missing
E	OK(at least three)	Missing	Missing
Fx	OK-	Missing/Incorrect	Missing/Incorrect
F	Missing/Incorrect	Missing/Incorrect	Missing/Incorrect

Table 1: Grading scheme

To get a Good, your program/answer must:

1. Completely meet the specification,e.g., produce correct output from example inputs (not too little or too much), or the like;
2. Be readable without overly commenting, be correctly indented, and use reasonable names (for variables, method, classes etc.);
3. Be well designed for the problem at hand

For every point above that is not satisfied, the program/answer takes one step down the grade ladder from Good down to Missing/Fail.