Lab session 9 – Lisp/Clojure (part II)

Unit	Programming languages: principles and design (6G6Z1110) Programming languages – SE frameworks (6G6Z1115)
Lecturer	Rob Frampton
Week	10
Portfolio element	This lab is not part of the portfolio.

Description

The aim with this lab exercises is to practise the several Lisp/Clojure elements as studied in the lecture "Lisp/Clojure – part II".

Note: This lab is meant to be performed on *repl.it* (https://repl.it/). On repl.it, you can create Clojure programs and compile them using a remote JVM. You can also create an account on *repl.it*, which is free, then select "*New Repl*" under the "*my repls*" menu and then, "*Clojure*" under the "*Practical*" programming languages section. Although the use of repl.it is very intuitive, take few minutes to get familiar with its GUI before you start with the lab exercises.

Exercises

Exercise 1

Write a function named `min-seq` which takes a vector of numbers as an argument, and returns the smallest element of the vector. You should use the `reduce` function and the `min` function to do this.

An example output of your function might be:

```
   (min-seq [ 3 1 4 3 ])
=> 1
```

Exercise 2

Write a function named `hasTenMultiple` which takes a vector of numbers as an argument, and returns `true` if there is at least one element which is a multiple of ten.

An example output of your function might be:

```
   (hasTenMultiple [ 11 23 49 ])
=> false
   (hasTenMultiple [ 11 23 40 ])
=> true
```

Exercise 3

Write a function named `filterStrings`, which takes a vector of strings as an argument. If the vector is empty, the function should print the message "*One or more strings must be provided*". Otherwise, it returns all the strings from the input whose length is greater than 5.

An example output of your function might be:

```
   (filterStrings [ ])
One or more strings must be provided
=> nil
   (filterStrings [ "Assume", "Ample", "Extract", "End" ])
=> ("Assume" "Extract")
```

Exercise 4

a) Write a function named `ten-divides` which takes an argument *n*. The function should return `true` if *n* is divisible by 10 and `false` otherwise. You will need to use the `mod` function.

b) Create a function named `random-tens` which takes no arguments. On the first line, define a variable named `numbers` using the `def` function, with the following value (This code generates 10 random numbers between 0 and 99):

```
(repeatedly 10 #(rand-int 100))
```

c) On the second line of your function, you will need to insert and complete the following if statement:

```
(if (not-any? <???>)
  <??? code for true>
  <??? code for false>
)
```

The `not-any?` function will return `true` if a given function returns `true` for **none** of the elements in a given sequence. Add the appropriate arguments so that this `if` function tests to see if none of the random numbers are divisible by 10. You will use the function you wrote in part a.

d) When none of the random numbers are divisible by 10, the function should print out "No multiples of ten" using the `println` function.

e) Otherwise, the function should return all the values which are multiples of 10. You will need to use the `filter` function.

An example output of your function might be:

```
   (random-tens)
No multiples of ten
=> nil
   (random-tens)
=> (40 30)
```

f) Write a function named `divides?` which takes as inputs a divisor *x* and a number *n*. The function should return `true` if *x* divides *n* and `false` otherwise. For example:

```
   (divides? 2 10)
=> true
   (divides? 10 13)
=> false
```

g) You will now modify your `random-tens` function so that it uses the more generic `divides?` function rather than `ten-divides`. However, because you are passing the function *itself* as an argument to other functions, it is not obvious how you can do this while fixing the first argument to `divides?` as 10. Instead, you will need to create an *anonymous function* which takes a single argument *x*, and calls `divides?` with 10 as the first argument and *x* as the second, which can now replace `ten-divides`.

Test that your function works the same as in part e).