Лабораторна робота №7.

Функціональне програмування на Clojure

Написати програми на мовах програмування Clojure, Lisp, F# для реалізації наступних завдань:

0.
Написати функцію генерації двох випадкових чисел з 1,2,3,...,14. Запустити її та отримані два числа будуть вашими варіантами завдань, які дані нижче. Скріншот результату додати до звіту. ***15 завдання виконують усі!*** В звіті порівняти часові характеристики виконування програм на різних мовах.

1.
Write a function which allows you to create function compositions. The parameter list should take a variable number of functions, and create a function that applies them from right-to-left.

```
(= [3 2 1] ((__ rest reverse) [1 2 3 4]))

(= 5 ((__ (partial + 3) second) [1 2 3 4]))

(= true ((__ zero? #(mod % 8) +) 3 5 7 9))

(= "HELLO" ((__ #(.toUpperCase %) #(apply str %) take) 5 "hello world"))
```

2.
Write a function which returns a sequence of lists of x items each. Lists of less than x items should not be returned.

```
(= (__ 3 (range 9)) '((0 1 2) (3 4 5) (6 7 8)))

(= (__ 2 (range 8)) '((0 1) (2 3) (4 5) (6 7)))

(= (__ 3 (range 8)) '((0 1 2) (3 4 5)))
```

3.
Take a set of functions and return a new function that takes a variable number of arguments and returns a sequence containing the result of applying each function left-to-right to the argument list.

```
(= [21 6 1] ((__ + max min) 2 3 5 1 6 4))

(= ["HELLO" 5] ((__ #(.toUpperCase %) count) "hello"))

(= [2 6 4] ((__ :a :c :b) {:a 2, :b 4, :c 6, :d 8 :e 10}))
```

4.
Write a function that splits a sentence up into a sorted list of words. Capitalization should not affect sort order and punctuation should be ignored.

```
(= (__  "Have a nice day.")

   ["a" "day" "Have" "nice"])

(= (__  "Clojure is a fun language!")

   ["a" "Clojure" "fun" "is" "language"])

(= (__  "Fools fall for foolish follies.")

   ["fall" "follies" "foolish" "Fools" "for"])
```

5.
Given a string of comma separated integers, write a function which returns a new comma separated string that only contains the numbers which are perfect squares.

```
(= (__ "4,5,6,7,8,9") "4,9")

(= (__ "15,16,25,36,37") "16,25,36")
```

6.
Clojure has many sequence types, which act in subtly different ways. The core functions typically convert them into a uniform "sequence" type and work with them that way, but it can be important to understand the behavioral and performance differences so that you know which kind is appropriate for your application.

Write a function which takes a collection and returns one of :map, :set, :list, or :vector - describing the type of collection it was given.
You won't be allowed to inspect their class or use the built-in predicates like list? - the point is to poke at them and understand their behavior.

```
(= :map (__ {:a 1, :b 2}))

(= :list (__ (range (rand-int 20))))

(= :vector (__ [1 2 3 4 5 6]))

(= :set (__ #{10 (rand-int 5)}))

(= [:map :set :vector :list] (map __ [{} #{} [] ()]))
```

7.
The trampoline function takes a function f and a variable number of parameters. Trampoline calls f with any parameters that were supplied. If f returns a function, trampoline calls that function with no arguments. This is repeated, until the return value is not a function, and then trampoline returns that non-function value. This is useful for implementing mutually recursive algorithms in a way that won't consume the stack.
test not run

```
(= __
```

```
(letfn

  [(foo [x y] #(bar (conj x y) y))

   (bar [x y] (if (> (last x) 10)

                x

                 #(foo x (+ 2 y))))]

  (trampoline foo [] 1)))
```

8.
Write a function which finds all the anagrams in a vector of words. A word x is an anagram of word y if all the letters in x can be rearranged in a different order to form y. Your function should return a set of sets, where each sub-set is a group of words which are anagrams of each other. Each sub-set should have at least two words. Words without any anagrams should not be included in the result.
test not run

```
(= (__ ["meat" "mat" "team" "mate" "eat"])

   #{#{"meat" "team" "mate"}})

(= (__ ["veer" "lake" "item" "kale" "mite" "ever"])

   #{#{"veer" "ever"} #{"lake" "kale"} #{"mite" "item"}})
```

9.
A balanced number is one whose component digits have the same sum on the left and right halves of the number. Write a function which accepts an integer n, and returns true iff n is balanced.

```
(= true (__ 11))

(= true (__ 121))

(= false (__ 123))

(= true (__ 0))

(= false (__ 88099))

(= true (__ 89098))

(= true (__ 89089))

(= (take 20 (filter __ (range)))

   [0 1 2 3 4 5 6 7 8 9 11 22 33 44 55 66 77 88 99 101])
```

10.
A function f defined on a domain D induces an equivalence relation on D, as follows: a is equivalent to b with respect to f if and only if (f a) is equal to (f b). Write a function with arguments f and D that computes the equivalence classes of D with respect to f.

test not run

```
(= (__ #(* % %) #{-2 -1 0 1 2})

   #{#{0} #{1 -1} #{2 -2}})


(= (__ #(rem % 3) #{0 1 2 3 4 5 })

   #{#{0 3} #{1 4} #{2 5}})


(= (__ identity #{0 1 2 3 4})

   #{#{0} #{1} #{2} #{3} #{4}})


(= (__ (constantly true) #{0 1 2 3 4})

   #{#{0 1 2 3 4}})
```

11.
Given an input sequence of keywords and numbers, create a map such that each key in the map is a keyword, and the value is a sequence of all the numbers (if any) between it and the next keyword in the sequence.
test not run

```
(= {} (__ []))

(= {:a [1]} (__ [:a 1]))

(= {:a [1], :b [2]} (__ [:a 1, :b 2]))

(= {:a [1 2 3], :b [], :c [4]} (__ [:a 1 2 3 :b :c 4]))
```

12.
Write a function which returns a sequence of digits of a non-negative number (first argument) in numerical system with an arbitrary base (second argument). Digits should be represented with their integer values, e.g. 15 would be [1 5] in base 10, [1 1 1 1] in base 2 and [15] in base 16.
test not run

```
(= [1 2 3 4 5 0 1] (__ 1234501 10))

(= [0] (__ 0 11))

(= [1 0 0 1] (__ 9 2))

(= [1 0] (let [n (rand-int 100000)](__ n n)))

(= [16 18 5 24 15 1] (__ Integer/MAX_VALUE 42))
```

13.
Write an oscillating iterate: a function that takes an initial value and a variable number of functions. It should return a lazy sequence of the functions applied to the value in order, restarting from the first function after it hits the end.

```
(= (take 3 (__ 3.14 int double)) [3.14 3 3.0])

(= (take 5 (__ 3 #(- % 3) #(+ 5 %))) [3 0 5 2 7])

(= (take 12 (__ 0 inc dec inc dec inc)) [0 1 0 1 0 1 2 1 2 1 2 3])
```

14.
Given a mathematical formula in prefix notation, return a function that calculates the value of the formula. The formula can contain nested calculations using the four basic mathematical operators, numeric constants, and symbols representing variables. The returned function has to accept a single parameter containing the map of variable names to their values.

```
(= 2 ((__ '(/ a b))

      '{b 8 a 16}))

(= 8 ((__ '(+ a b 2))

      '{a 2 b 4}))

(= [6 0 -4]

     (map (__ '(* (+ 2 a)

                  (- 10 b)))

            '[{a 1 b 8}

              {b 5 a -2}

              {a 2 b 11}]))

(= 1 ((__ '(/ (+ x 2)

              (* 3 (+ y 1))))

      '{x 4 y 1}))
```

15.
In what follows, m, n, s, t denote nonnegative integers, f denotes a function that accepts two arguments and is defined for all nonnegative integers in both arguments.

In mathematics, the function f can be interpreted as an infinite matrix with infinitely many rows and columns that, when written, looks like an ordinary matrix but its rows and columns cannot be written down completely, so are terminated with ellipses. In Clojure, such infinite matrix can be represented as an infinite lazy sequence of infinite lazy sequences, where the inner sequences

represent rows.

Write a function that accepts 1, 3 and 5 arguments

with the argument f, it returns the infinite matrix A that has the entry in the i-th row and the j-th column equal to f(i,j) for i,j = 0,1,2,...;
with the arguments f, m, n, it returns the infinite matrix B that equals the remainder of the matrix A after the removal of the first m rows and the first n columns;
with the arguments f, m, n, s, t, it returns the finite s-by-t matrix that consists of the first t entries of each of the first s rows of the matrix B or, equivalently, that consists of the first s entries of each of the first t columns of the matrix B.

```
(= (take 5 (map #(take 6 %) (__ str)))

   [["00" "01" "02" "03" "04" "05"]

    ["10" "11" "12" "13" "14" "15"]

    ["20" "21" "22" "23" "24" "25"]

    ["30" "31" "32" "33" "34" "35"]

    ["40" "41" "42" "43" "44" "45"]])

(= (take 6 (map #(take 5 %) (__ str 3 2)))

   [["32" "33" "34" "35" "36"]

    ["42" "43" "44" "45" "46"]

    ["52" "53" "54" "55" "56"]

    ["62" "63" "64" "65" "66"]

    ["72" "73" "74" "75" "76"]

    ["82" "83" "84" "85" "86"]])

(= (__ * 3 5 5 7)

   [[15 18 21 24 27 30 33]

    [20 24 28 32 36 40 44]

    [25 30 35 40 45 50 55]

    [30 36 42 48 54 60 66]

    [35 42 49 56 63 70 77]])

(= (__ #(/ % (inc %2)) 1 0 6 4)

   [[1/1 1/2 1/3 1/4]

    [2/1 2/2 2/3 1/2]
```

```
    [3/1 3/2 3/3 3/4]
    [4/1 4/2 4/3 4/4]
    [5/1 5/2 5/3 5/4]
    [6/1 6/2 6/3 6/4]])
(= (class (__ (juxt bit-or bit-xor)))
   (class (__ (juxt quot mod) 13 21))
   (class (lazy-seq)))
(= (class (nth (__ (constantly 10946)) 34))
   (class (nth (__ (constantly 0) 5 8) 55))
   (class (lazy-seq)))
(= (let [m 377 n 610 w 987
         check (fn [f s] (every? true? (map-indexed f s)))
         row (take w (nth (__ vector) m))
         column (take w (map first (__ vector m n)))
         diagonal (map-indexed #(nth %2 %) (__ vector m n w w))]
     (and (check #(= %2 [m %]) row)
          (check #(= %2 [(+ m %) n]) column)
          (check #(= %2 [(+ m %) (+ n %)]) diagonal)))
   true)
```