Final CAP

Curs 2024-25 (14-I-2025) Duració: 2 hores

1.-
a/ (1 punt) Definiu una funció `number-of-rotations` que donat un vector ordenat però rotat a la dreta un cert nombre de vegades, retorni quants cops ha estat rotat aquest vector.
Per exemple:

```
(number-of-rotations [10 2 4 6])  →  1
(number-of-rotations [7 8 10 2 3 4 5 6])  →  3
(number-of-rotations [2 3 4 5 6])  →  0
```

Nota: Podeu fer servir la funció `.indexOf`, que donat un vector i un valor, `(.indexOf vector valor)` retorna l'índex on aquest valor apareix per primer cop en el vector, o -1 si el valor no hi és dins el vector.

b/ (1 punt) Definiu una funció `fsmap` que, donats un element `x` i una llista `fs` de funcions, fa que `(fsmap x fs)` retorni una llista amb les aplicacions successives de les funcions de `fs` a `x`. És a dir, la llista resultant ha de contenir `x`, seguit de `(f1 x)`, `(f2 (f1 x))`, etc., on `f1`, `f2`, ... són les funcions de la llista `fs`. A més, cal definir `fsmap` fent servir `reductions`.
Per exemple:

```
(fsmap 3 [(fn [x] (+ 2 x)) (fn [x] (* 3 x)) (fn [x] (+ 4 x))]) → (3 5 15 19)
```

Solució:

a/

```
(defn number-of-rotations [rotated-vec]
  (let [min-val (apply min rotated-vec)
        min-index (.indexOf rotated-vec min-val)]
    min-index))
```

b/

```
(defn fsmap [a fs]
  (reductions (fn [acc f] (f acc)) a fs))
```

2.- (2.5 punts) Sigui la funció `(move n from to aux)` que retorna la seqüència dels moviments que cal fer per resoldre el problema de les *Torres de Hanoi* (el suposem conegut, en altre cas veure apèndix) per a n anelles o discos.

La funció `(move n from to aux)` és:

```
(defn move [n from to aux]
  (if (= n 1)
    [{:from from :to to}]
    (concat
      (move (dec n) from aux to)
      [{:from from :to to}]
      (move (dec n) aux to from))))
```

Si definim `(defn torres-de-hanoi [n]`
`(move n "A" "C" "B"))`

i executem `(torres-de-hanoi 3)` obtenim la seqüència (*pretty printed*):

```
({:from "A", :to "C"}
 {:from "A", :to "B"}
 {:from "C", :to "B"}
 {:from "A", :to "C"}
 {:from "B", :to "A"}
 {:from "B", :to "C"}
 {:from "A", :to "C"})
```

Es demana que transformeu aquesta funció a recursiva final fent servir CPS, anomenem-la `move-cps`, i després feu servir `move-cps` com a funció auxiliar per poder fer servir `trampoline` i definir la funció "*trampolinitzada*" `move-t` (l'execució de la qual consumeix una quantitat constant de pila d'execució).

Solució:

```
(defn move-cps [n from to aux k]
  (if (= n 1)
    (k [{:from from :to to}])
    (move-cps (dec n) from aux to
              (fn [moves1]
                (move-cps (dec n) aux to from
                          (fn [moves2]
                            (k (concat moves1 [{:from from :to to}] moves2))))))))

(defn towers-of-hanoi-cps [n]
  (move-cps n "A" "C" "B" identity))

(defn move-t [nn f t a]
  (letfn [(move-cps-t [n from to aux k]
            (if (= n 1)
              #(k [{:from from :to to}])
              #(move-cps-t (dec n) from aux to
                          (fn [moves1]
                            (fn [] (move-cps-t (dec n) aux to from
                                       (fn [moves2]
                                         (fn [] (k (concat moves1 [{:from from :to to}] moves2))))))))))]
    (trampoline move-cps-t nn f t a identity)))

(defn towers-of-hanoi-t [n]
  (move-t n "A" "C" "B"))
```

3.- (2.5 punts) Recordeu la definició circular de llistes "*infinites*" que vam veure a classes de laboratori? Per exemple: `(def naturals (lazy-seq (cons 0 (map inc naturals))))` o bé `(def factorials (lazy-seq (cons 1N (map * factorials (iterate inc 1N)))))`

Ara suposem que tenim una funció `f` que serveix per definir una seqüència a_0, a_1, a_2,... on

$$a_0 = 1$$
$$a_{n+1} = f(a_n)$$

Així doncs, la seqüència és a_0, $f(a_0)$, $f(f(a_0))$, ...

Definiu ara una llista "*infinita*", anomenem-la `seq-general`, amb els elements de la seqüència a_0, a_1, a_2,... fent servir aquesta tècnica de la definició *circular*. És a dir, cal que definiu `seq-general` tal que la seva definició tingui la forma:

`(def seq-general ...`*`transformació, que inclou f, de la mateixa llista`* `seq-general)`

Solució:

```
(def seq-general (cons 1 (lazy-seq (map f seq-general))))
```

4.- (3 punts) Implementeu la funció `aplicacio-condicional`, una funció que té dues funcions com a paràmetres, `f` i `condicio`. `f` és una funció que pren dos arguments, i `condicio` és una funció-predicat que acceptarà un sol argument i retornarà `true` o `false`.

`aplicacio-condicional` retorna una funció `fr` tal que:

```
                                 |sí ---> retorna gr
fr(x) --és cert (condicio x)?---|
                                 |no ---> Ignorem x i retornem fr (la mateixa fr)


                                 |sí ---> retorna (f x y)
gr(y) --és cert (condicio y)?---|
                                 |no ---> Ignorem y i retornem gr (la mateixa gr)
```

Així, podem passar tants arguments com calgui fins que n'hi hagi dos que satisfan condicio:

```
(def suma-si-parell (aplicacio-condicional + even?))

((suma-si-parell 2) 4) → 6
(((((suma-si-parell 2) 3) 5) 7) 6) → 8
```

obviament el darrer argument que fem servir cal que satisfaci la condició:

```
((((((suma-si-parell 2) 3) 5) 7) 6) 9) →
Execution error (ClassCastException) at user/eval147 (REPL:1).
class java.lang.Long cannot be cast to class clojure.lang.IFn (java.lang.Long is
in module java.base of loader 'bootstrap'; clojure.lang.IFn is in unnamed module
of loader 'app')
```

En altre cas intentaria aplicar un nombre com si fos una funció.

Solució:

```
(defn aplicacio-condicional
  "f és una funció de dos paràmetres, i condició és un predicat"
  [f condicio]
  (letfn [(g [x] (letfn [(h [y]
                           (if (condicio y) (f x y) h))]
                  (if (condicio x) h g)))]
    g))
```

Apèndix:

Les Torres de Hanoi

- Les torres de Hanoi és un problema matemàtic inventat el 1883 pel matemàtic francès François Édouard Anatole Lucas (1842-1891).
- Tenim **tres pals** i **N anelles**, totes de mides diferents.
- Inicialment tenim totes les anelles al primer dels pals, apilades en ordre de mida decreixent:

- L'objectiu és moure totes les anelles al tercer pal (igualment apilades):

- En cada pas només podem moure l'anella al cim d'una pila d'anelles.
- Cal moure-la a un pal buit, o a un pal amb l'anella del cim és més gran que l'anella que volem moure.

Estratègia per resoldre Les Torres de Hanoi

- **Cas base**: Un joc amb una sola anella (N=1). Trivial de resoldre.
- **Cas recursiu**: La intuició fonamental aquí és veure que: Per moure N anelles del pal inicial a un pal final, podem primer moure N-1 anelles del pal inicial al pal intermedi,

- moure l'anella que queda al pal inicial, al pal final,

- i després moure les N-1 anelles del pal intermedi al pal final.