CAP - Problemes

Jordi Delgado, Gerard Escudero,

Exercicis diversos

Exercicis

Disclaimer:

- Aquests problemes no estan agrupats seguint el temari (tret del que demani l'enunciat, que sí pot deixar clar que un problema pertany a un tema o a un altre). Estan pensats per, un cop s'acabi el curs, poder practicar tot el que heu aprés sobre programació funcional en Clojure.

- Tampoc estan ordenats per grau de dificultat. Els problemes del principi no són més fàcils que els del final. De fet, aquesta llista anirà creixent i no reordenarem el material cada cop que afegim problemes nous (que esperem que sigui sovint). N'hi ha de molt fàcils, quasi-trivials, i n'hi ha de més difícils. Tots són, pensem, factibles.

- Els problemes no estan resolts. Naturalment els professors tenim una o vàries solucions de cada problema, però no les farem públiques. Si teniu problemes per resoldre algun exercici pregunteu, que per a això estem els professors (entre d'altres coses).

- De fet, aquesta llista no hauria de durar gaire, ja que esperem que en un futur no llunyà acabi forman part dels problemes de Clojure al Jutge. Així que tindrà, de ben segur, una vida limitada.

Exercicis

- Per què passa això?:

```
(fn? range) 👉 true
(fn? (first '(range 10))) 👉 false
```

- Escriu una funció `(concat-elements a-seq)` que pren una seqüència de seqüències i les concatena juntament amb `concat`. No utilitzeu `apply` per implementar aquesta funció.

```
(concat-elements [])                👉 ()
(concat-elements [[:a :b]])         👉 (:a :b)
(concat-elements [[10 20] [30 40]]) 👉 (10 20 30 40)
```

- Escriu una funció `(str-cat a-seq)` que pren una seqüència d'*strings* i les junta afegint un caràcter *espai* entre elles. Feu servir `reduce`, tot i que potser a l'*string* buida li cal un tractament especial.

```
(str-cat ["Sóc" "del" "Barça"])   👉 "Sóc del Barça"
(str-cat ["Ya" "si" "eso"])       👉 "Ya si eso"
(str-cat ["quants" " " "espais"]) 👉 "quants   espais"
(str-cat [])                      👉 ""
```

Exercicis

- Escriu una funció `(interposar x a-seq)` que posa `x` entre cada element de la seqüència `a-seq`. Recordeu com funciona `conj` per a vectors.

```
(interposar 0 [1 2 3])           👉 (1 0 2 0 3)
(interposar "," ["Yo "mi" "me"]) 👉 ("Yo" "," "mi" "," "me")
(interposar :a [1])              👉 (1)
(interposar :a [])               👉 ()
```

- Escriu una funció `(paritat a-seq)` que posa en un conjunt tots els elements d'`a-seq` que apareixen un nombre senar de vegades.

```
(paritat [:a :b :c])    👉 #{:a :b :c}
(paritat [:a :a :b :b]) 👉 #{}
(paritat [1 2 3 1])     👉 #{2 3}
```

- Escriu una funció `(pred-and p1 p2 ... pN)` que admet $N \geq 0$ predicats com a arguments i retorna una funció, diguem-ne `result`, tal que `(result x)` és `true` si i només si és `true` per a tots i cada un dels predicats, és a dir, `(p1 x)` 👉 `true`, `(p2 x)` 👉 `true`,...,`(pN x)` 👉 `true`. En altre cas `result` ha de retornar `false`. Si $N = 0$, la funció que retorna ha de ser la funció constant `true`.

Exercicis

- Coneixeu la funció de Clojure interleave? Escriu una funció que inverteix el que fa `interleave`. Donada una seqüència `a-seq` i un nombre `n`, `(inverteix-interleave a-seq n)` retorna una seqüència amb `n` seqüències tals que l'aplicació d'`interleave` sobre aquestes seqüències resultaria en `a-seq`

```
;; Com tenim que:
(interleave '(1 3 5) '(2 4 6))                  👉 (1 2 3 4 5 6)
(interleave '(0 3 6) '(1 4 7) '(2 5 8))         👉 (0 1 2 3 4 5 6 7 8)
(interleave '(0 5) '(1 6) '(2 7) '(3 8) '(4 9)) 👉 (0 1 2 3 4 5 6 7 8 9)

;; aleshores...
(inverteix-interleave [1 2 3 4 5 6] 2) 👉 ((1 3 5) (2 4 6))
(inverteix-interleave (range 9) 3)     👉 ((0 3 6) (1 4 7) (2 5 8))
(inverteix-interleave (range 10) 5)    👉 ((0 5) (1 6) (2 7) (3 8) (4 9))
```

o, dit d'una altra manera:

```
(apply interleave (inverteix-interleave a-seq n)) ≡ (seq a-seq)
```

Exercicis

- Escriu una funció `(my-partition n a-seq)` que, donats un nombre `n` i una seqüència `a-seq`, retorna una seqüència de llistes amb `n` elements d'`a-seq` cada una. Les llistes de menys d'`n` elements s'ignoren. Naturalment, no podeu fer servir `partition` ni `partition-all`.

```
(my-partition 3 (range 9)) 👉 ((0 1 2) (3 4 5) (6 7 8))
(my-partition 2 (range 8)) 👉 ((0 1) (2 3) (4 5) (6 7))
(my-partition 3 (range 9)) 👉 ((0 1 2) (3 4 5))
```

- Escriu una funció `(my-frequencies coll)` que donada una col·lecció `coll`, retorni un diccionari on cada entrada té com a clau un element de `coll` i com a valor el nombre d'aparicions d'aquest element a la col·lecció. No podeu fer servir `frequencies`.

```
(my-frequencies [1 1 2 3 2 1 1])      👉 {1 4, 2 2, 3 1}
(my-frequencies [:b :a :b :a :b])     👉 {:a 2, :b 3}
(my-frequencies '([1 2] [1 3] [1 3])) 👉 {[1 2] 1, [1 3] 2}
```

- Implementeu la funció `reductions`, sense fer servir `reductions` de Clojure, és clar. La podeu anomenar, per ser originals, `my-reductions`. Tingueu en compte la *laziness*.

Exercicis

- Donades diverses funcions com a arguments, feu una funció `my-juxt` que retorni una funció que, donat un nombre variable d'arguments, retorni la seqüència resultant d'aplicar les funcions, d'esquerra a dreta, als arguments. No podeu fer servir `juxt`.

```
((my-juxt + max min) 2 3 5 1 6 4) 👉 (21 6 1)
((my-juxt :a :c :b) {:a 2, :b 4, :c 6, :d 8 :e 10}) 👉 (2 6 4)
```

- Escriu una funció `merge-maps` que tingui com a paràmetres una funció `f` i un nombre variable de diccionaris. La vostra funció hauria de retornar un diccionari que consti de la resta de diccionaris *conj-ed* al primer. Si es troba la mateixa clau en més d'un diccionari, el valor de la clau cal modificar-lo fent servir la funció `f`: `(f valor-actual valor-trobat)`. Els diccionaris es processaran d'esquerra a dreta. No podeu fer servir `merge-with`.

```
(merge-maps * {:a 2, :b 3, :c 4} {:a 2} {:b 2} {:c 5}) 👉 {:a 4, :b 6, :c 20}
(merge-maps - {1 10, 2 20} {1 3, 2 10, 3 15}) 👉 {1 7, 2 10, 3 15}
(merge-maps concat {:a [3], :b [6]} {:a [4 5], :c [8 9]} {:b [7]})
👉 {:a [3 4 5], :b [6 7], :c [8 9]}
```

Exercicis

- Escriu una funció `anagrames` que trobi tots els anagrames d'un vector de paraules. Una paraula x és un anagrama de la paraula y si totes les lletres de x es poden reordenar en un ordre diferent per formar y. La vostra funció hauria de retornar un conjunt de conjunts, on cada subconjunt és un grup de paraules que són anagrames entre si. Cada subconjunt ha de tenir almenys dues paraules. Les paraules sense anagrama no s'han d'incloure al resultat.

```
(anagrames ["val" "jo" "pila" "lav"]) 👉 #{#{"lav" "val"}}
(anagrames ["astro" "nas" "sant" "ostra" "san"])
👉 #{#{"san" "nas"} #{"astro" "ostra"}}
(anagrames ["llac" "sopa" "itres" "call" "estri" "nou" "isert" "onu" ])
👉 #{#{"nou" "onu"} #{"itres" "estri" "isert"} #{"call" "llac"}}
```

- Escriu una funció `aplana-parcial` que aplani qualsevol combinació imbricada de col·leccions seqüencials (llistes, vectors, etc.), però mantingui els elements seqüencials de nivell més baix. El resultat hauria de ser una seqüència de seqüències amb només un nivell d'imbricació.

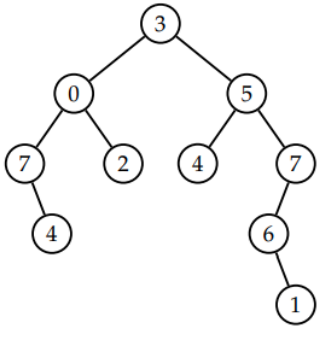
```
(aplana-parcial [["Fes"] ["Res"]]) 👉 (["Fes"] ["Res"])
(aplana-parcial [[[[:a :b]]] [[:c :d]] [:e :f]]) 👉 ([:a :b] [:c :d] [:e :f])
(aplana-parcial '((1 2)((3 4)((((5 6 7)))))))) 👉 ((1 2) (3 4) (5 6 7))
```

Exercicis

- Torneu a la plana 46 del Tema 1 i re-implementeu `balancejat` d'una manera més funcional.
- Penseu un **Quine** en Clojure. Al tanto, que no es fàcil, tot i que és molt més fàcil en Clojure que en, per exemple, C, C++ o Java. Podeu trobar solucions *on-line* (al tanto, proveu-les perquè algunes són incorrectes), però busqueu-les només després d'haver-ho provat de debó.

Exercicis

- [X30150] Feu una funció `decodifica-arbre-binari-int` que donada una codificació *size-based* d'un arbre binari, retorni l'arbre binari expressat en termes de diccionaris de la forma `{:val n, :L <arbre esq.>, :R <arbre dre.>}`. Aquesta codificació està explicada al problema X30150 del Jutge. Per exemple, si tenim l'arbre



amb codificació `[10 3 4 0 2 7 0 4 0 2 0 5 1 4 0 7 2 6 0 1 0]`, aleshores:

```
(decodifica-arbre-binari-int 10 3 4 0 2 7 0 4 0 2 0 5 1 4 0 7 2 6 0 1 0) 👉
;; resultat "pretty-printed"
{:val 3, :L {:val 0, :L {:val 7, :L nil,
                                 :R {:val 4, :L nil, :R nil}},
                         :R {:val 2, :L nil, :R nil}},
          :R {:val 5, :L {:val 4, :L nil, :R nil},
                         :R {:val 7, :L {:val 6, :L nil,
                                                 :R {:val 1, :L nil, :R nil}},
                                     :R nil}}}
```

Fonts dels exercicis

La gran majoria dels problemes llistats no ens els hem inventat nosaltres. Les fonts d'on hem tret alguns problemes són, de moment, les següents:

- Curs Functional Programming in Clojure. Els autors del curs són Juhana Laurinharju, Jani Rahkola i Ilmari Vacklin. El curs va ser un MOOC que es va fer des del Departament de *Computer Science* a l'Universitat de Helsinki. Web.

- Col·lecció de problemes 4ever-Clojure. Lloc amb més de cent problemes pensats per ser resolts en Clojure, amb solucions (si us plau, no mireu les solucions fins no haver pensat força un problema; altrament és temps perdut).

- El Jutge, que no necessita presentació.