Parcial CAP

Curs 2024-25 (5-XI-2024) **Duració: 2 hores**

1.- (1 punt) Per quina raó cal el **`letfn`**? No en tenim prou amb el **`let`** per definir funcions locals? O, dit d'una altra manera, què puc fer amb el **`letfn`** que no puc fer amb el **`let`**?

Solució:

El lligam entre símbols i el valor de les expressions en el **`let`** es fa seqüencialment, de manera que en les expressions que apareixen dins el **`let`** puc fer referència a símbols que han estat lligats pel **`let`** *anteriorment*. En aquest cas seria impossible definir, per exemple, funcions mútuament recursives.

Això no passa amb el **`letfn`**. Amb el **`letfn`** sí que podem definir funcions mútuament recursives. Per exemple:

```
(defn even-or-odd? [n]
  (letfn [(my-even? [n]
            (if (zero? n) "even"
                (my-odd? (dec n))))
          (my-odd? [n]
            (if (zero? n) "odd"
                (my-even? (dec n))))]
    (my-even? n)))
```

2.- (2 punts) Suposem que fem una versió del `take-while`:

```
(defn my-take-while [f s]
  (if (empty? s) '()
      (let [[cap & cua] s]
        (if (f cap)
          (cons cap (my-take-while f cua))
          '()))))
```

Demostra (o argumenta rigorosament) que la funció **`(partial my-take-while f)`** és en realitat un **`fold`**, i escriu una versió de **`my-take-while`** fent servir **`fold`**.

Solució:

Observant **`my-take-while`** ens fixem que **`(partial my-take-while f)`** té la forma que la premisa de la propietat universal de **`fold`** requereix.

Així, aplicant aquesta propietat, tenim que:

```
(partial my-take-while f) =
(partial fold #(if (f %1) (cons %1 %2) '()) '())
```

I ara podem escriure:

```
(defn my-take-while' [f s]
  ((partial fold #(if (f %1) (cons %1 %2) '()) '()) s))
```

3.- (2 punts) Definiu una versió de la funció **`first`**, anomeneu-la per exemple **`my-first`**, fent servir **`fold`**.

Solució:

```
(def my-first (partial fold (fn [x y] x) nil))
```

En realitat, no importa el que posem com a valor inicial. Hem fet servir **`nil`**, però podríem haver posat qualsevol cosa.

4.- En Haskell tenim una funció **`until`** que, donat un predicat **`p`**, una funció **`f`** i un valor inicial **`v`**, va aplicant la funció **`f`** tal que **`v`**, **`f(v)`**, **`f(f(v))`**,... fins que es satisfa el predicat. Per exemple: **`(until #(> % 100) #(* 2 %) 1)`** té com a resultat 128

a) **(1 punt)** Implementeu la funció **`until`** en Clojure utilitzant el **`recur`**.

Solució:

```
(defn until [pred f x]
    (loop [v x]
        (if (pred v)
            v
            (recur (f v)))))
```

b) **(1 punt)** Implementeu la funció **`until`** en Clojure amb funcions d'ordre superior.

Solució:

```
(defn until [p f v]
    (first (drop-while (comp not p) (iterate f v))))
```

c) **(1 punt)** Implementeu l'algorisme d'Euclides per calcular el màxim comú divisor utilitzant la funció **`until`**:

```
Entrada: a, b
1. Si a=b, mcd=a, fi.
2. Si a>b, canvi de a per a-b, anar a 1.
3. Si a<b, canvi de b per b-a, anar a 1.
```

Solució:

```
(defn mcd [a b]
    (first
        (until #(apply = %)
               #(let [[a, b] %]
                   (if (< a b)
                     (list a (- b a))
                     (list (- a b) b)))
                (list a b)))))
```

5.- (2 punts) Un vector es considera especial si cada parell dels seus elements adjacents conté dos nombres amb paritat diferent. Feu una funció **`vector-especial?`** que, donat un vector d'enters **`nums`**, retorna **`True`** si **`nums`** és un vector especial, en cas contrari, retorna **`False`**. No podeu fer servir ni recursivitat ni `loop/recur`.

Exemples:

```
(vector-especial? [1])       => True
(vector-especial? [2,1,4])   => True
(vector-especial? [4,3,1,6]) => False
```

ja que `(3,1)` són senars els dos

Solució:

```
(defn vector-especial? [nums]
  (->> nums
       (#(map (fn [x y] (list x y)) % (rest %)))
       (map #(apply + %))   ;; parell + senar = senar
       (every? odd?)))
```