Principles of Programming Languages, 2017.07.05

Notes
- Total available time: 2h.
- You may use any written material you need, and write in Italian, if you prefer.
- You cannot use electronic devices during the exam.

Exercise 1, Scheme (10 pts)

1) Define a mutable binary tree data structure, using structs.
2) Define a destructive map operation for binary trees, called tmap!.
E.g. **(define t1 (branch (branch (leaf 1)(leaf 2))(leaf 3)));** after **(map! (lambda (x) (+ x 1)) t1)**, t1 becomes **(branch (branch (leaf 2)(leaf 3))(leaf 4))**
3) Define a destructive reverse, called reverse!, which takes a binary tree and keeps it structure, but "reversing" all the values in the leaves. E.g. **(reverse! t1)** makes t1 the tree **(branch (branch (leaf 3)(leaf 2))(leaf 1))**.

Exercise 2, Haskell (12 pts)

Consider the following binary tree data structure:

data Tree a = Nil | Leaf a | Branch (Tree a)(Tree a) deriving (Show, Eq)

1) Define a **tcompose** operation, which takes a function f and two trees, t1 and t2, and returns a tree with the same structure as t1, but with leaves replaced by subtrees having the same structure of t2: each leaf is obtained by applying f to the value stored in the previous leaf, and the corresponding value in t2.

E.g. **t1 = Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3), t2 = Branch (Leaf 6) (Leaf 7)**

tcompose (+) t1 t2 is

Branch (Branch (Branch (Leaf 7) (Leaf 8)) (Branch (Leaf 8) (Leaf 9))) (Branch (Leaf 9) (Leaf 10))

2) Define a purely functional (i.e. non destructive) version of the reverse presented in Es. 1.3.

E.g. **revtree t1** is **Branch (Branch (Leaf 3) (Leaf 2)) (Leaf 1).**

Exercise 3, Erlang (10 pts)

Consider a binary tree stored with tuples, e.g. **{branch, {branch, {leaf, 1}, {leaf, 2}}, {leaf, 3}}**.

Define an **activate** function, which takes a binary tree and a binary function **f** and creates a network of actors having the same structure of the given tree. Actors corresponding to leaves run a function called **leafy**, that must be defined, which answer to the message **{ask, P}** by sending to process P the pair **{self(), V}**, where V is the value stored in the leaf, then terminate.

Actors for the branches run a function called **branchy**, that must be also defined: if they receive an **{ask, P}** request like that of leaves, they ask both their sons; when they receive the answers, they call f on the obtained values, then send the result V to P as **{self(), V}** and terminate.

E.g. running the following code:

```
test() ->
   T1 = {branch, {branch, {leaf, 1}, {leaf, 2}}, {leaf, 3}},
   A1 = activate(T1, fun min/2),
   A1 ! {ask, self()},
   receive
      {A1, V} -> V
   end.
```

should return 1.

Solutions

Es 1

```
(struct branch ((left  #:mutable)
                (right #:mutable)))

(struct leaf ((val #:mutable)))

(define (tmap! f t)
  (if (leaf? t)
      (set-leaf-val! t (f (leaf-val t)))
      (begin
        (tmap! f (branch-left t))
        (tmap! f (branch-right t)))))

(define (reverse-visit t)
  (if (leaf? t)
      (list (leaf-val t))
      (begin
        (append
         (reverse-visit (branch-right t))
         (reverse-visit (branch-left t))))))

(define (reverse! t)
  (let ((vals (reverse-visit t)))
    (tmap!
     (lambda (x)
       (let ((v (car vals)))
         (set! vals (cdr vals))
         v))
     t)))
```

Es 2

I assume Tree to be an instance of Functor and Foldable, as seen in class, to use folds and fmap.

```
tcompose :: (a -> b -> c) -> Tree a -> Tree b -> Tree c
tcompose f Nil _ = Nil
tcompose f (Leaf x) t1 = fmap (f x) t1
tcompose f (Branch l r) t1 = Branch (tcompose f l t1) (tcompose f r t1)

revtree t = t1
  where (t1, _) = revtree' t $ foldl (\x y -> y:x) [] t
        revtree' Nil xs = (Nil, xs)
        revtree' (Leaf v) (x:xs) = (Leaf x, xs)
        revtree' (Branch l r) xs = let (l', xs')  = revtree' l xs
                                       (r', xs'') = revtree' r xs'
                                   in (Branch l' r', xs'')
```

Es 3

```
leafy(V) ->
    receive
        {ask, Pid} -> Pid ! {self(), V}
    end.

branchy(L, R, Fun, Ready, Dad) ->
    receive
        {ask, Pid} ->
            L ! {ask, self()},
            R ! {ask, self()},
            branchy(L, R, Fun, Ready, Pid);
        {L, V1} ->
           case Ready of
               true  -> Dad ! {self(), Fun(V1, R)};
               false -> branchy(V1, R, Fun, true, Dad)
           end;
        {R, V1} ->
           case Ready of
               true  -> Dad ! {self(), Fun(L, V1)};
               false -> branchy(L, V1, Fun, true, Dad)
           end
    end.

activate({leaf, X}, _) ->
    spawn(?MODULE, leafy, [X]);
activate({branch, L, R}, Fun) ->
    spawn(?MODULE, branchy, [activate(L, Fun), activate(R, Fun), Fun, false, none]).
```