Principles of Programming Languages, 2021.06.22

Important notes
- Total available time: 1h 40'.
- You may use any written material you need, and write in Italian, if you prefer.
- You cannot use electronic devices during the exam: every phone must be turned off and kept on your table.
- You cannot use library functions not covered in class in your code.

Exercise 1, Scheme (10 pts)

Define a function *mix* which takes a variable number of arguments $x_0\ x_1\ x_2\ ...\ x_n$, the first one a function, and returns the list $(x_1\ (x_2\ ...\ (x_0(x_1)\ x_0(x_2)\ ...\ x_0(x_n))\ x_n)\ x_{n-1})\ ...\ x_1)$.

E.g.

```
(mix (lambda (x) (* x x)) 1 2 3 4 5)
```

returns: '(1 (2 (3 (4 (5 (1 4 9 16 25) 5) 4) 3) 2) 1)

Exercise 2, Haskell (11 pts)

Define a data-type called BTT which implements trees that can be binary or ternary, and where every node contains a value, but the empty tree (Nil). Note: there must not be unary nodes, like leaves.

1) Make *BTT* an instance of Functor and Foldable.

2) Define a concatenation for *BTT*, with the following constraints:

- If one of the operands is a binary node, such node must become ternary, and the other operand will become the added subtree (e.g. if the binary node is the left operand, the rightmost node of the new ternary node will be the right operand).
- If both the operands are ternary nodes, the right operand must be appened on the right of the left operand, by recursively calling concatenation.

3) Make *BTT* an instance of Applicative.

Exercise 3, Erlang (11 pts)

Create a function *node_wait* that implements nodes in a tree-like topology. Each node, which is a separate agent, keeps track of its parent and children (which can be zero or more), and contains a value. An integer weight is associated to each edge between parent and child.
A node waits for two kind of messages:

- *{register_child, ...}*, which adds a new child to the node (replace the dots with appropriate values),
- *{get_distance, Value}*, which causes the recipient to search for *Value* among its children, by interacting with them through appropriate messages. When the value is found, the recipient answers with a message containing the minimum distance between it and the node containing "Value", considering the sum of the weights of the edges to be traversed. If the value is not found, the recipient answers with an appropriate message. While a node is searching for a value among its children, it may not accept any new children registrations. E.g., if we send *{get_distance, a}* to the root process, it answers with the minimum distance between the root and the closest node containing the atom *a* (which is 0 if *a* is in the root).

Solutions

Es 1

```
(define (f g . L)
  (foldr (lambda (x y)
           (list x y x))
         (map g L)
         L))
```

Es 2

```
data BTT a = Nil | B a (BTT a)(BTT a) | T a (BTT a)(BTT a)(BTT a) deriving (Eq, Show)

instance Functor BTT where
  fmap f Nil = Nil
  fmap f (B a l r) = B (f a)(fmap f l)(fmap f r)
  fmap f (T a l c r) = T (f a)(fmap f l)(fmap f c)(fmap f r)

instance Foldable BTT where
  foldr f i Nil = i
  foldr f i (B x l r) = f x $ foldr f (foldr f i r) l
  foldr f i (T x l c r) = f x $ foldr f (foldr f (foldr f i r) c) l

x <++> Nil = x
Nil <++> x = x
(B x l r) <++> y = (T x l r y)
y <++> (B x l r) = (T x y l r)
(T x l c r) <++> v@(T x' l' c' r') = (T x l c (r <++> v))


ltconcat t = foldr (<++>) Nil t
ltconcmap f t = ltconcat $ fmap f t

instance Applicative BTT where
  pure x = (B x Nil Nil)
  x <*> y = ltconcmap (\f -> fmap f y) x
```

Es 3

```
node_wait(Parent, Elem, Children) ->
    receive
        {register_child, Child, Weight} ->
            node_wait(Parent, Elem, [{Child, Weight} | Children]);
        {get_distance, Value} -> if
                                     Value == Elem ->
                                         Parent ! {distance, Value, self(), 0},
                                         node_wait(Parent, Elem, Children);
                                     true ->
                                         node_comp_dist(Parent, Elem, Children, Value)
                                 end
    end.

node_comp_dist(Parent, Elem, Children, Value) ->
    [Child ! {get_distance, Value} || {Child, _} <- Children],
    Dists = [receive
                 {distance, Value, Child, D} ->
                     D + Weight;
                 {not_found, Value, Child} ->
                     not_found
             end || {Child, Weight} <- Children],
    FoundDists = lists:filter(fun erlang:is_integer/1, Dists),
    case FoundDists of
        [] ->
            Parent ! {not_found, Value, self()};
        _ ->
            Parent ! {distance, Value, self(), lists:min(FoundDists)}
    end,
    node_wait(Parent, Elem, Children).
```