Principles of Programming Languages, 2023.06.12

Important notes
- Total available time: 2h.
- You may use any written material you need, and write in English or in Italian.
- 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)

Write a function, called *fold-left-right*, that computes both *fold-left* and *fold-right,* returning them in a pair. Very important: the implementation must be *one-pass*, for efficiency reasons, i.e. it must consider each element of the input list only once; hence it is not correct to just call Scheme's fold-left and -right.

Example: `(fold-left-right string-append "" '("a" "b" "c"))` is the pair `("cba" . "abc")`.

Exercise 2, Haskell (11 pts)

Define a *partitioned list* data structure, called Part, storing three elements:

1. a *pivot* value,
2. a list of elements that are all less than or equal to the pivot, and
3. a list of all the other elements.

Implement the following utility functions, writing their types:

- *checkpart*, which takes a Part and returns true if it is valid, false otherwise;
- *part2list*, which takes a Part and returns a list of all the elements in it;
- *list2part*, which takes a pivot value and a list, and returns a Part;

Make Part an instance of Foldable and Functor, if possible. If not, explain why.

Exercise 3, Erlang (11 pts)

Consider the following implementation of *mergesort* and write a parallel version of it.

```
mergesort([L]) -> [L];
mergesort(L) ->
    {L1, L2} = lists:split(length(L) div 2, L),
    merge(mergesort(L1), mergesort(L2)).

merge(L1, L2) -> merge(L1, L2, []).
merge([], L2, A) -> A ++ L2;
merge(L1, [], A) -> A ++ L1;
merge([H1|T1], [H2|T2], A) when H2 >= H1 -> merge(T1, [H2|T2], A ++ [H1]);
merge([H1|T1], [H2|T2], A) when H1 > H2  -> merge([H1|T1], T2, A ++ [H2]).
```

Note: multichance students do not need to solve Exercise 1.

Solutions

Ex 1

```
(define (fold-left-right f i l)
  (let loop ((left i)
             (right (lambda (x) x))
             (xs l))
    (if (null? xs)
        (cons left (right i))
        (loop (f (car xs) left)
              (lambda (x)
                (right (f (car xs) x)))
              (cdr xs)))))
```

Ex 2

```
data Part a = Part [a] a [a] deriving Show

checkpart :: Ord a => Part a -> Bool
checkpart (Part x a y) = null (filter (\v -> (v > a)) x) &&
                         null (filter (\v -> (v <= a)) y)

part2list :: Part a -> [a]
part2list (Part x a y) = x ++ [a] ++ y

list2part :: (Ord a) => a -> [a] -> Part a
list2part a l = l2ph a l [] [] where
  l2ph a [] l r = Part l a r
  l2ph a (x:xs) l r | x <= a = l2ph a xs (x:l) r
  l2ph a (x:xs) l r = l2ph a xs l (x:r)

Foldable is easy:
instance Foldable Part where
  foldr f i p = foldr f i (part2list p)

Functor: A trivial implementation like the following

instance Functor Part where
  fmap f (Part x a y) = Part (fmap f x) (f a) (fmap f y)

is not correct, because if we take e.g.
p1 = Part [1,2,3] 4 [5,6,6]; p2 = fmap (10 -) p1
p2 is not a correct partition. We could use list2part to fix the solution, but this requires that, if f :: (a -> b), b must be
an instance of Ord.
```

Ex 3

```
mergesortp(L) ->
    Me = self(),
    msp(Me, L),
    receive
        {Me, R} -> R
    end.

msp(Pid, [L]) ->
    Pid ! {self(), [L]};
msp(Pid, L)  ->
    {L1, L2} = lists:split(length(L) div 2, L),
    Me = self(),
    Pid1 = spawn(fun() -> msp(Me, L1) end),
    Pid2 = spawn(fun() -> msp(Me, L2) end),
    receive
        {Pid1, Sorted1} ->
            receive
                {Pid2, Sorted2} ->
                    Pid ! {self(), merge(Sorted1, Sorted2)}
            end
    end.
```