Principles of Programming Languages, 2020.01.15

Important 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: every phone must be turned off and on the table.
- You cannot use library functions not covered in class in your code.

Exercise 1, Scheme (11 pts)

Consider the *Foldable* and *Applicative* type classes in Haskell. We want to implement something analogous in Scheme for *vectors*. Note: you can use the following library functions in your code: *vector-map, vector-append*.

1) Define *vector-foldl* and *vector-foldr*.

2) Define *vector-pure* and *vector-<*>*.

Exercise 2, Haskell (12 pts)

The following data structure represents a cash register. As it should be clear from the two accessor functions, the first component represents the current item, while the second component is used to store the price (not necessarily of the item: it could be used for the total).

```
data CashRegister a = CashRegister { getReceipt :: (a, Float) } deriving (Show, Eq)

getCurrentItem = fst . getReceipt

getPrice = snd . getReceipt
```

1) Make CashRegister an instance of Functor and Applicative.

2) Make CashRegister an instance of Monad.

Exercise 3, Erlang (9 pts)

We want to implement something like Python's *range* in Erlang, using processes.

E.g.

```
R = range(1,5,1)      % starting value, end value, step

next(R)               % is 1

next(R)               % is 2

…

next(R)               % is 5

next(R)               % is the atom stop_iteration
```

Define *range* and *next*, where *range* creates a process that manages the iteration, and *next* a function that talks with it, asking the current value.

Solutions

Es 1

```
(define (vector-foldr f i v)
  (let loop ((cur (- (vector-length v) 1))
             (out i))
    (if (< cur 0)
        out
        (loop (- cur 1) (f (vector-ref v cur) out)))))

(define (vector-foldl f i v)
  (let loop ((cur 0)
             (out i))
    (if (>= cur (vector-length v))
        out
        (loop (+ cur 1) (f (vector-ref v cur) out)))))

(define (vector-concat-map f v)
  (vector-foldr vector-append #() (vector-map f v)))

(define vector-pure vector)

(define (vector-<*> fs xs)
  (vector-concat-map (lambda (f) (vector-map f xs)) fs))
```

Es 2

```
instance Functor CashRegister where
  fmap f cr = CashRegister (f $ getCurrentItem cr, getPrice cr)

instance Applicative CashRegister where
  pure x = CashRegister (x, 0.0)
  CashRegister (f, pf) <*> CashRegister (x, px) = CashRegister (f x, pf + px)

instance Monad CashRegister where
  CashRegister (oldItem, price) >>= f = let newReceipt = f oldItem
                                        in CashRegister (getCurrentItem newReceipt, price + (getPrice
newReceipt))
```

Es 3

```
ranger(Current, Stop, Inc) ->
    receive
        {Pid, next} ->
            if
                Current =:= Stop -> Pid ! stop_iteration;
                true -> Pid ! Current,
                        ranger(Current + Inc, Stop, Inc)
            end
    end.

range(Start, Stop, Inc) ->
    spawn(?MODULE, ranger, [Start, Stop, Inc]).

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