Principles of Programming Languages, 2022.06.16

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 kept on your table.
- You cannot use library functions not covered in class in your code.

Exercise 1, Scheme (10 pts)

Define a *list-to-compose* pure function, which takes a list containing functions of one argument and returns their composition.

E.g. *(list-to-compose (list f g h))* is the function *f(g(h(x))*.

Exercise 2, Haskell (12 pts)

Consider the "fancy pair" data type (called *Fpair*), which encodes a pair of the same type *a*, and may optionally have another component of some "showable" type *b*, e.g. the character '$'.

Define *Fpair*, parametric with respect to both *a* and *b*.

1) Make *Fpair* an instance of Show, where the implementation of show of a fancy pair e.g. encoding *(x, y, '$')* must return the string "[**x**$**y**]", where **x** is the string representation of *x* and **y** of *y*. If the third component is not available, the standard representation is "[**x**, **y**]".

2) Make *Fpair* an instance of *Eq* — of course the component of type *b* does not influence the actual value, being only part of the representation, so pairs with different representations could be equal.

3) Make *Fpair* an instance of Functor, Applicative and Foldable.

Exercise 3, Erlang (10 pts)

Define a program *tripart* which takes a list, two values x and y, with x < y, and three functions, taking one argument which must be a list.

- *tripart* first partitions the list in three sublists, one containing values that are less than both x and y, one containing values v such that $x \leq v \leq y$, and one containing values that are greater than both x and y.
- Three processes are then spawned in parallel, running the three given functions and passing the three sublists in order (i.e. the first function must work on the first sublist and so on).
- Lastly, the program must wait the termination of the three processes in the spawning order, assuming that each one will return the pair {P, V}, where P is its PID and V the resulting value.
- *tripart* must return the three resulting values in a list, with the resulting values in the same order as the corresponding sublists.

Solutions

Es 1

```
(define (list-to-compose lst)
  (lambda (x)
    (foldr (lambda (y acc)
             (y acc)) x lst)))
```

Es 2

```
data Fpair s a = Fpair a a s | Pair a a

instance (Show a, Show s) => Show (Fpair s a) where
  show (Fpair x y t) = "[" ++ (show x) ++ (show t) ++ (show y) ++ "]"
  show (Pair x y) = "[" ++ (show x) ++ ", " ++ (show y) ++ "]"

simplify (Fpair x y _) = (x, y)
simplify (Pair x y) = (x, y)

instance (Eq a) => Eq (Fpair s a) where
  x == y = (simplify x) == (simplify y)

instance Functor (Fpair s) where
  fmap f (Fpair x y t) = (Fpair (f x) (f y) t)
  fmap f (Pair x y) = (Pair (f x) (f y))

instance Applicative (Fpair s) where
  pure x = (Pair x x)
  (Fpair f g _) <*> (Fpair x y v) = Fpair (f x) (g y) v
  (Pair f g) <*> (Fpair x y v) = Fpair (f x) (g y) v
  (Fpair f g v) <*> (Pair x y) = Fpair (f x) (g y) v
  (Pair f g) <*> (Pair x y) = Pair (f x) (g y)

instance Foldable (Fpair s) where
  foldr f i (Fpair x y _) = (f x (f y i))
  foldr f i (Pair x y) = (f x (f y i))
```

Es 3

```
helper([X|L],P1,P2,L1,L2,L3) when (X < P1)  and (X < P2)  -> helper(L,P1,P2,[X|L1],L2,L3);
helper([X|L],P1,P2,L1,L2,L3) when (X >= P1) and (X =< P2) -> helper(L,P1,P2,L1,[X|L2],L3);
helper([X|L],P1,P2,L1,L2,L3) when (X > P1)  and (X > P2)  -> helper(L,P1,P2,L1,L2,[X|L3]);
helper([],_,_,L1,L2,L3) -> [L1,L2,L3].

part(L,X,Y) -> helper(L,X,Y,[],[],[]).

tripart(L,X,Y,F1,F2,F3) ->
    [D1,D2,D3] = part(L,X,Y),
    P1 = spawn(?MODULE, F1, [D1]),
    P2 = spawn(?MODULE, F2, [D2]),
    P3 = spawn(?MODULE, F3, [D3]),
    receive
        {P1, V1} ->
            receive
                {P2, V2} ->
                    receive
                        {P3, V3} ->
                            [V1,V2,V3]
                    end
            end
    end.
```