Principles of Programming Languages, 2023.07.03

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 (11 pts)

Define a *let*** construct that behaves like the standard *let**, but gives to variables provided without a binding the value of the last defined variable. It also contains a default value, stated by a special keyword *def:*, to be used if the first variable is given without binding.

For example:

```
(let** def: #f
       (a (b 1) (c (+ b 1)) d (e (+ d 1)) f)
       (list a b c d e f))
```

should return *'(#f 1 2 2 3 3)*, because *a* assumes the default value *#f*, while *d* = *c* and *f* = *e*.

Exercise 2, Haskell (11 pts)

1. Define a data structure, called D2L, to store lists of possibly depth two, e.g. like [1,2,[3,4],5,[6]].
2. Implement a *flatten* function which takes a D2L and returns a flat list containing all the stored values in it in the same order.
3. Make D2L an instance of Functor, Foldable, Applicative.

Exercise 3, Erlang (11 pts)

1. Define a "deep reverse" function, which takes a "deep" list, i.e. a list containing possibly lists of any depths, and returns its reverse.
   E.g. *deeprev([1,2,[3,[4,5]],[6]])* is *[[6],[[5,4],3],2,1]*.
2. Define a parallel version of the previous function.

Note: multichance students do not need to solve Exercise 3.

Solutions

Ex 1

```
(define-syntax let**
  (syntax-rules (def:)
    ((_  def: v (var) istr ...)
     ((lambda (var) istr ...)
      v))
    ((_ def: v ((var val)) istr ...)
     ((lambda (var) istr ...)
      val))
    ((_  def: v ((var val) . rest) istr ...)
     ((lambda (var)
        (let** def: val rest istr ...))
      val))
    ((_  def: v (var . rest) istr ...)
     ((lambda (var)
        (let** def: v rest istr ...))
      v))))
```

Ex 2

```
data D2L a = D2Nil | D2Cons1 a (D2L a) | D2Cons2 [a] (D2L a) deriving (Show, Eq)

flatten D2Nil = []
flatten (D2Cons1 x xs) = (x : flatten xs)
flatten (D2Cons2 xs ys) = xs ++ flatten ys

instance Functor D2L where
  fmap f D2Nil = D2Nil
  fmap f (D2Cons1 x xs) = D2Cons1 (f x) (fmap f xs)
  fmap f (D2Cons2 xs ys) = D2Cons2 (fmap f xs) (fmap f ys)

instance Foldable D2L where
  foldr f i D2Nil = i
  foldr f i (D2Cons1 x xs) = f x (foldr f i xs)
  foldr f i (D2Cons2 xs ys) = (foldr f (foldr f i ys) xs)

D2Nil +++ t = t
t +++ D2Nil = t
(D2Cons1 x xs) +++ t = D2Cons1 x (xs +++ t)
(D2Cons2 xs ys) +++ t = D2Cons2 xs (ys +++ t)

instance Applicative D2L where
  pure x = D2Cons1 x D2Nil
  fs <*> xs = foldr (+++) D2Nil (fmap (\f -> fmap f xs) fs)
```

Ex 3

```
deeprev([]) -> [];
deeprev([X|Xs]) -> V = deeprev(X),
                   Vs = deeprev(Xs),
                   Vs ++ [V];
deeprev(X) -> X.

deeprevp(L) ->
    P = self(),
    dp(P, L),
    receive
        {P, R} -> R
    end.

dp(Pid, []) -> Pid ! {self(), []};
dp(Pid, [X|Xs]) ->
    Self = self(),
    P1 = spawn(fun() -> dp(Self, X) end),
    P2 = spawn(fun() -> dp(Self, Xs) end),
    receive
        {P1, V} ->
            receive
                {P2, Vs} ->
                    Pid ! {Self, Vs ++ [V]}
            end
    end;
dp(Pid, X) -> Pid ! {self(), X}.
```