Principles of Programming Languages, 2018.07.20

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.

Exercise 1, Scheme (12 pts)

1) Give a purely functional definition of *fep*, which takes a list $(x_1\ x_2\ ...\ x_n)$ and returns $(x_1\ (x_2\ (...\ (x_n\ (x_1\ x_2\ ...\ x_n)\ x_n)\ x_{n-1})\ ...)\ x_1)$.

2) Consider the following code; explain how it works, and what is the output of the call *(run)*.

```
(define saved '())

(define (push-k x)
  (set! saved (append saved (list x))))

(define (poprun-k)
  (if (null? saved)
      #f
      (let ((x (car saved)))
        (set! saved (cdr saved))
        (x))))
```

```
(define (c1 x)
  (call/cc (lambda (k)
             (push-k k)))
  (set! x (+ x 1))
  (display "c1 ")(displayln x))

(define (c2 y)
  (call/cc (lambda (k)
             (push-k k)))
  (set! y (* y 2))
  (display "c2 ")(displayln y))

(define (run)
  (c1 0) (c2 2) (poprun-k))
```

Exercise 2, Haskell (12 pts)

1) Consider the function *fep* of Exercise 1. We want to implement an Haskell version of it, but of course we cannot use plain lists: explain why and define a datatype (say *DeepList*) for it.

2) Make *DeepList* an instance of Show, such that its representation is like that of Scheme.

3) Implement *fep*.

4) Make *DeepList* an instance of Functor.

Exercise 3, Erlang (8 pts)

Consider the following Erlang program:

```
buffer(Content) ->
    receive
        {get, From} ->
            if
                Content =:= [] ->
                    From ! empty,
                    buffer([]);
                true ->
                    [H|T] = Content,
                    From ! H,
                    buffer(T)
            end;
        {put, Data} ->
            buffer(Content ++ [Data])
    end.

producer(From, To, Buffer, Father) ->
    if
        From < To ->
            Buffer ! {put, From},
            io:format("~w produced ~p~n", [self(), From]),
            producer(From+1, To, Buffer, Father);
        true -> Father ! {self(), done}
    end.
```

```
consumer(Buffer) ->
    Buffer ! {get, self()},
    receive
        empty ->
            io:format("~w: empty buffer~n", [self()]),
            consumer(Buffer);
        V ->
            io:format("~w consumed ~p~n", [self(), V]),
            consumer(Buffer)
    end.

main() ->
    B  = spawn_link(?MODULE, buffer, [[]]),
    P1 = spawn(?MODULE, producer, [0,10,B,self()]),
    C1 = spawn_link(?MODULE, consumer, [B]),
    C2 = spawn_link(?MODULE, consumer, [B]),
    receive
        {P1, done} -> exit(die)
    end.
```

Fix the system to have two producers, a more graceful exit, and to avoid links.

Solutions

Es 1

```
(define (deepena L)
  (foldr (lambda (x y)
          (list x y x))
         L
         L))
```

```
c1 1
c2 4
c1 2
c2 4
c2 8
c2 8
```

Es 2

```
data DeepList a = Val a | DeepList [DeepList a] deriving Eq

instance (Show a) => Show (DeepList a) where
  show (Val x) = " " ++ show x ++ " "
  show (DeepList ls) = "(" ++ (concatMap show ls) ++ ")"

infixl 1 -++-    -- concatenation
(DeepList xs) -++- (DeepList ys) = DeepList (xs ++ ys)

fep dl = fep' dl dl where
  fep' (DeepList []) z = z
  fep' (DeepList (x:xs)) z = (DeepList [x]) -++- DeepList [(fep' (DeepList xs) z)] -++- (DeepList [x])

instance Functor DeepList where
  fmap f (Val a) = Val $ f a
  fmap f (DeepList xs) = DeepList $ map (\x -> let (Val y) = x in Val (f y)) xs
```

Es 3

Producer is unchanged; buffer and consumer add, as a first clause in their receive, the following code:

```
        stop -> ok;
```

The main function is changed as follows:

```
main() ->
   B  = spawn(?MODULE, buffer,   [[]]),
   P1 = spawn(?MODULE, producer, [0,10,B,self()]),
   P2 = spawn(?MODULE, producer, [11,20,B,self()]), % an example new producer
   C1 = spawn(?MODULE, consumer, [B]),
   C2 = spawn(?MODULE, consumer, [B]),
   receive
      {P1, done} -> ok
   end,
   receive
      {P2, done} -> ok
   end,
   C1 ! stop,
   C2 ! stop,
   B  ! stop.
```