Principles of Programming Languages, 2018.07.06

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 (10 pts)

1) Give a purely functional definition of *deepen*, which takes a list (x_1 x_2 ... x_n) and returns ((... ((x_1) x_2) ...) x_n).

2) Write the construct *define-with-return:*, which takes a name *m*, used as a *return* function, a list function name + parameters, and a function body, and defines a function with the same characteristics, where calls to *m* are used to return a value.

E.g. if we define

(define-with-return: return (f x) ; note that the function name is f, while return is used, of course, for returning

(define a 12)

(return (+ x a))

'unreachable),

a call *(f 3)* should give 15.

Exercise 2, Haskell (12 pts)

Consider this datatype: *data Blob a = Blob a (a -> a)*

Note: in this exercise, do not consider the practical meaning of Blob; the only constraint is to use all the available data, and the types must be right!

E.g.

instance Show a => Show (Blob a) where

show (Blob x f) = "Blob " ++ (show (f x))

1) Can Blob automatically derive *Eq*? Explain how, why, and, if the answer is negative, make it an instance of *Eq*.

2) Make Blob an instance of the following classes: *Functor*, *Foldable*, and *Applicative.*

Exercise 3, Erlang (10 pts)

Consider the following program, containing some errors:

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

(see back)

```
producer(From, To, Buffer) ->
    if
        From < To ->
            Buffer ! {put, From},
            io:format("~w produced ~p~n", [self(), From]),
            producer(From+1, To, Buffer);
        true -> Buffer ! {put, done}
    end.
consumer(Buffer) ->
    Buffer ! {get, self()},
    receive
        done -> ok;
        V ->
            io:format("~w consumed ~p~n", [self(), V]),
            consumer(Buffer)
    end.
main() ->
    B = spawn(?MODULE, buffer, [[]]),
    P1 = spawn(?MODULE, producer, [1,10,B]),
    C1 = spawn(?MODULE, consumer, [B]),
    C2 = spawn(?MODULE, consumer, [B]).
```

1) Describe how it works;

2) write why and where it is broken;

3) fix it.

Solutions

Es 1

```
(define (deepen L)
  (foldl (lambda (x y)
           (list y x))
         (list (car L))
         (cdr L)))

(define-syntax define-with-return:
  (syntax-rules ()
    ((_ return (fun p1 ...) e1 ...)
     (define (fun p1 ...)
       (call/cc (lambda (return)
                  e1 ...))))))
```

Es 2

```
instance Eq a => Eq (Blob a) where
  (Blob x f) == (Blob y g) = (f x) == (g y)

instance Functor Blob where
  fmap f (Blob x g) = Blob (f (g x)) id

instance Foldable Blob where
  foldr f z (Blob x g) = f (g x) z

instance Applicative Blob where
  pure x = Blob x id
  (Blob fx fg) <*> (Blob x g) = Blob (((fg fx) . g) x) id
```

Es 3

The first error is *[H|T] = Content*, because the buffer could be empty, so this could crash it.
The second error is the approach to stop the system: there is only one producer and it produces “done” before ending. But this ends only one consumer, so both the buffer and the other consumer remain active.

A simple fix: the buffer sends a message “empty”, and the consumer works correspondingly. For the second error, a very rough fix is to link processes and kill them all when the producer is done. Of course there are many other more elegant but slightly more complex approaches.

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

producerfix(From, To, Buffer, Father) ->
```

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

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

mainfix() ->
    B  = spawn_link(?MODULE, bufferfix, [[]]), % two consumers
    P1 = spawn(?MODULE, producerfix, [1,10,B,self()]),
    C1 = spawn_link(?MODULE, consumerfix, [B]),
    C2 = spawn_link(?MODULE, consumerfix, [B]),
    receive
        {P1, doom} -> exit(die)
    end.
```