# *Principles of Programming Languages, 2017.07.20*

**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)

Consider the following code:

```
(define (print+sub x y)
  (display x)
  (display " ")
  (display y)
  (display " -> ")
  (- x y))

(define (puzzle)
  (call/cc (lambda (exit)
             (define (local e)
               (call/cc
                (lambda (local-exit)
                  (exit (print+sub e
                                   (call/cc
                                    (lambda (new-exit)
                                      (set! exit new-exit)
                                      (local-exit #f))))))))
             (local 6)
             (exit 2))))
```

1) Describe how it works.
2) What is the output of the evaluation of `(define x (puzzle))`? What is the value of x?
3) Can you see problems with this code?

## Exercise 2, Haskell (12 pts)

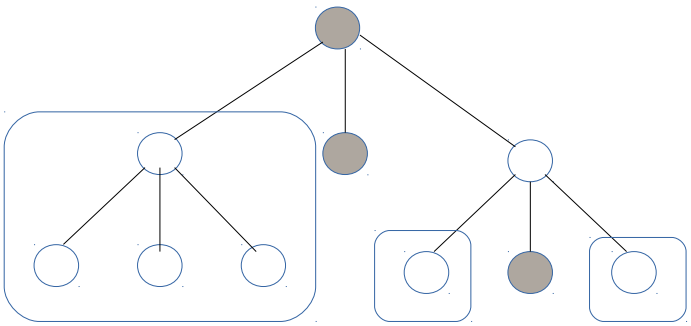
1) Define a ternary tree data structure, called Ttree, in which every node contain both a **value** and a **color**, which can be either yellow or blue. You can assume that a tree cannot be empty.

2) Make Ttree an instance of Functor.

3) Make Ttree an instance of Foldable.

4) Define a function yellowSubTrees, which returns a list containing all the maximal subtrees of a given Ttree that are all made of yellow nodes.

E.g. in the case of the following tree, where yellow nodes are depicted in white, the returned list contains the three outlined subtrees.



## Exercise 3, Erlang (8 pts)

We want to define a “dynamic list” data structure, where each element of the list is an actor storing a value. Such value can be of course read and set, and each get/set operation on the list can be performed in parallel.

1) Define **create_dlist**, which takes a number n and returns a dynamic list of length n. You can assume that each element store the value 0 at start.

2) Define the function **dlist_to_list**, which takes a dynamic list and returns a list of the contained values.

2) Define a **map** for dynamic list. Of course this operation has side effects, since it changes the content of the list.

# Solutions

Es 1
The code should set continuations such that the evaluated expression is the following:
(exit (print+sub 6 (new-exit 2)))
hence it prints "6 2 ->" and sets x to 4.

But this code wrongly assumes that the expression representing the function (the first exit call in this case) is evaluated before its parameters. In the current implementation of Racket it is so, so this works. But if we use another implementation of Scheme, the expression could become:

```
(new-exit (print+sub 6 (new-exit 2)))
```

So this prints "6 2 ->" then returns to the internal new-exit with the value 4 (6-2), printing "6 4 ->". Then returns to the internal new-exit with 2 (6-4), and so on, looping indefinitely.
To fix this problem we can modify puzzle like this:

```
(define (puzzle1)
  (call/cc (lambda (exit)
             (define old-exit exit)
             (define (local e)
               (call/cc
                (lambda (local-exit)
                  (old-exit (print+sub e
                                       (call/cc
                                        (lambda (new-exit)
                                          (set! exit new-exit)
                                          (local-exit #f))))))))
             (local 6)
             (exit 2))))
```

Es 2

```
data Color = Yellow | Blue deriving (Show, Eq)
data Ttree a = Tlf Color a | Tbr Color a (Ttree a) (Ttree a) (Ttree a) deriving (Show, Eq)

instance Functor Ttree where
  fmap f (Tlf c a) = Tlf c $ f a
  fmap f (Tbr c a l m r) = Tbr c (f a) (fmap f l) (fmap f m) (fmap f r)

instance Foldable Ttree where
  foldr f v (Tlf c a) = f a v
  foldr f v (Tbr c a l m r) =
    let v1 = foldr f v r
        v2 = foldr f v1 m
        v3 = foldr f v2 l
    in f a v3

getYellow x@(Tlf Yellow _) = ([x], True)
getYellow (Tlf Blue _) = ([], False)
getYellow x@(Tbr c _ l m r) =
  let chs = [l, m, r]
      ys = map getYellow chs
  in if c == Yellow && (and $ map (\(x,y) -> y) ys)
     then ([x], True)
     else (concat $ filter (/= []) $ map fst ys, False)

yellowSubtrees t = fst $ getYellow t
```

Es 3

```
cell(Value) ->
    receive
        {set, V} ->
            cell(V);
        {get, Pid} ->
            Pid ! Value,
            cell(Value)
    end.

delement_get(Element) ->
    Element ! {get, self()},
    receive
        V -> V
    end.

delement_set(Element, New) ->
    Element ! {set, New}.

create_dlist(0) -> [];
create_dlist(Size) -> [spawn(?MODULE, cell, [0]) | create_dlist(Size-1)].

dlist_map([], _) -> ok;
dlist_map([X|Xs], Fun) ->
    delement_set(X, Fun(delement_get(X))),
    dlist_map(Xs, Fun).
```