# *Principles of Programming Languages, 2020.02.07*

**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 (11 pts)

Implement this new construct: (**each-until** *var* **in** *list* **until** *pred* **:** *body*), where keywords are written in boldface. It works like a for-each with variable *var*, but it can end before finishing all the elements of *list* when the predicate *pred* on *var* becomes true.

E.g.

```
(each-until x in '(1 2 3 4)
            until (> x 3) :
            (display (* x 3))
            (display " "))
```

shows on the screen: 3 6 9

## Exercise 2, Haskell (11 pts)

Consider a data type *PriceList* that represents a list of items, where each item is associated with a price, of type Float:

```
data PriceList a = PriceList [(a, Float)]
```

1) Make *PriceList* an instance of Functor and Foldable.

2) Make *PriceList* an instance of Applicative, with the constraint that each application of a function in the left hand side of a <*> must increment a right hand side value's price by the price associated with the function.

E.g.

```
PriceList [(("nice "++), 0.5), (("good "++), 0.4)]  <*>
PriceList [("pen", 4.5), ("pencil", 2.8), ("rubber", 0.8)]
```

is

```
PriceList [("nice pen",5.0),("nice pencil",3.3),("nice rubber",1.3),("good pen",4.9),
("good pencil",3.2),("good rubber",1.2)]
```

## Exercise 3, Erlang (11 pts)

We want to create a simplified implementation of the "Reduce" part of the MapReduce paradigm. To this end, define a process "reduce_manager" that keeps track of a pool of reducers. When it is created, it stores a user-defined associative binary function `ReduceF`. It receives messages of the form `{reduce, Key, Value}`, and forwards them to a different "reducer" process for each key, which is created lazily (i.e. only when needed). Each reducer serves requests for a unique key.
Reducers keep into an accumulator variable the result of the application of `ReduceF` to the values they receive. When they receive a new value, they apply `ReduceF` to the accumulator and the new value, updating the former. When the reduce_manager receives the message `print_results`, it makes all its reducers print their key and incremental result.

(see back)

For example, the following code (where the meaning of *string:split* should be clear from the context):

```
word_count(Text) ->
    RMPid = start_reduce_mgr(fun (X, Y) -> X + Y end),
    lists:foreach(fun (Word) -> RMPid ! {reduce, Word, 1} end, string:split(Text, " ", all)),
    RMPid ! print_results,
    ok.
```

causes the following print:

```
1> mapreduce:word_count("sopra la panca la capra campa sotto la panca la capra crepa").
sopra: 1
la: 4
panca: 2
capra: 2
campa: 1
sotto: 1
crepa: 1
ok
```

# Solutions

Es 1

```
(define-syntax each-until
  (syntax-rules (in until :)
    ((_ x in L until pred : body ...)
       (let loop ((xs L))
         (unless (null? xs)
           (let ((x (car xs)))
             (unless pred
               (begin
                 body ...
                 (loop (cdr xs))))))))))
```

Es 2

```
pmap :: (a -> b) -> Float -> PriceList a -> PriceList b
pmap f v (PriceList prices) = PriceList $ fmap (\x -> let (a, p) = x
                                                      in (f a, p+v)) prices
instance Functor PriceList where
  fmap f prices = pmap f 0.0 prices

instance Foldable PriceList where
  foldr f i (PriceList prices) = foldr (\x y -> let (a, p) = x
                                           in  f a y) i prices

(PriceList x) +.+ (PriceList y) = PriceList $ x ++ y

plconcat x = foldr (+.+) (PriceList []) x

instance Applicative PriceList where
  pure x = PriceList [(x, 0.0)]
  (PriceList fs) <*> xs = plconcat (fmap (\ff -> let (f, v) = ff
                                                 in pmap f v xs) fs)
```

Es 3

```
start_reduce_mgr(ReduceF) ->
    spawn(?MODULE, reduce_mgr, [ReduceF, #{}]).

reduce_mgr(ReduceF, Reducers) ->
    receive
         print_results ->
             lists:foreach(fun ({_, RPid}) -> RPid ! print_results end, maps:to_list(Reducers));
         {reduce, Key, Value} ->
             case Reducers of
                  #{Key := RPid} ->
                      RPid ! {Key, Value},
                      reduce_mgr(ReduceF, Reducers);
                  _ ->
                      NewReducer = spawn(?MODULE, reducer, [ReduceF, Key, Value]),
                      reduce_mgr(ReduceF, Reducers#{Key => NewReducer})
             end
    end.

reducer(ReduceF, Key, Result) ->
    receive
         print_results ->
             io:format("~s: ~w~n", [Key, Result]);
         {Key, Value} ->
             reducer(ReduceF, Key, ReduceF(Result, Value))
    end.
```