Principles of Programming Languages, 2022.09.01

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 <u>turned off</u> and kept on your table.
- You cannot use library functions not covered in class in your code.

Exercise 1, Scheme (10 pts)

We want to implement a version of call/cc, called *store-cc*, where the continuation is called only once and it is implicit, i.e. we do not need to pass a variable to the construct to store it. Instead, to run the continuation, we can use the associated construct *run-cc* (which may take parameters). The composition of *store-cc* must be managed using in the standard last-in-first-out approach.

E.g. if we run:

```
(define (test)
  (define x 0)
  (store-cc
    (displayln "here")
    (set! x (+ 1 x)))
  (displayln x)
  (set! x (+ 1 x))
  x)
(test)
```

we will get:

```
here
1
2
```

and if we call `(run-cc)`

we get:

```
2
3
```

and the continuation is discarded.

Exercise 2, Haskell (10 pts)

We want to implement a binary tree where in each node is stored data, together with the number of nodes contained in the subtree of which the current node is root.

1. Define the data structure.
2. Make it an instance of Functor, Foldable, and Applicative.

Exercise 3, Erlang (12 pts)

We want to implement a parallel foldl, *parfold(F, L, N)*, where the binary operator F is associative, and N is the number of parallel processes in which to split the evaluation of the fold. Being F associative, parfold can evaluate foldl on the N partitions of L in parallel. Notice that there is no starting (or accumulating) value, differently from the standard foldl.
You may use the following libray functions:

lists:foldl(<function>, <starting value>, <list>)

lists:sublist(<list>, <init>, <length>), which returns the sublist of <list> starting at position <init> and of length <length>, where the first position of a list is 1.

Solutions

Es 1

```
(define *stored-cc* '())

(define-syntax store-cc
  (syntax-rules ()
    ((_ e ...)
     (call/cc (lambda (k)
                (set! *stored-cc* (cons k *stored-cc*))
                e ...)))))

(define (run-cc . v)
  (let ((k (car *stored-cc*)))
    (set! *stored-cc* (cdr *stored-cc*))
    (apply k v)))
```

Es 2

```
data Ctree a = Cnil | Ctree a Int (Ctree a) (Ctree a) deriving (Show, Eq)

cvalue Cnil = 0
cvalue (Ctree _ x _ _) = x

cnode x t1 t2 = Ctree x ((cvalue t1) + (cvalue t2) + 1) t1 t2
cleaf x = cnode x Cnil Cnil

instance Functor Ctree where
    fmap f Cnil = Cnil
    fmap f (Ctree v c t1 t2) = Ctree (f v) c (fmap f t1)(fmap f t2)

instance Foldable Ctree where
  foldr f i Cnil = i
  foldr f i (Ctree x _ t1 t2) = f x $ foldr f (foldr f i t2) t1

x +++ Cnil = x
Cnil +++ x = x
(Ctree x v t1 t2) +++ t =  cnode x t1 (t2 +++ t)

ttconcat = foldr (+++) Cnil
ttconcmap f t = ttconcat $ fmap f t

instance Applicative Ctree where
  pure = cleaf
  x <*> y = ttconcmap (\f -> fmap f y) x
```

Es 3

```
partition(L, N) ->
    M = length(L),
    Chunk = M div N,
    End = M - Chunk*(N-1),
    parthelp(L, N, 1, Chunk, End, []).

parthelp(L, 1, P, _, E, Res) ->
    Res ++ [lists:sublist(L, P, E)];
parthelp(L, N, P, C, E, Res) ->
    R = lists:sublist(L, P, C),
    parthelp(L, N-1, P+C, C, E, Res ++ [R]).

parfold(F, L, N) ->
    Ls = partition(L, N),
    W = [spawn(?MODULE, dofold, [self(), F, X]) || X <- Ls],
    [R|Rs] = [receive {P, V} -> V end || P <- W],
    lists:foldl(F, R, Rs).

dofold(Proc, F, [X|Xs]) ->
    Proc ! {self(), lists:foldl(F, X, Xs)}.
```