Principles of Programming Languages, 2019.01.16

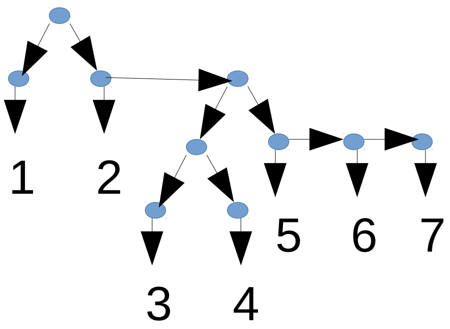
Notes
- Total available time: 1h 40’.
- You may use any written material you need, and write in Italian, if you prefer.
- You cannot use electronic devices during the exam.

Exercise 1, Scheme (8 pts)

Define a pure function *f* with a variable number of arguments, that, when called like *(f x_1 x_2 .. x_n)*, returns: *(x_n (x_{n-1} (.. (x_1 (x_n x_{n-1} .. x_1))..)*. Function *f* must be defined using only fold operations for loops.

Exercise 2, Haskell (14 pts)

We want to define a data structure, called Listree, to define structures working both as lists and as binary trees, like in the next figure.



1) Define a datatype for Listree.

2) Write the example of the figure with the defined data structure.

3) Make Listree an instance of Functor.

4) Make Listree an instance of Foldable.

5) Make Listree an instance of Applicative.

Exercise 3, Erlang (9 pts)

Define a process P, having a local *behavior* (a function), that answer to three commands:

- load is used to load a new function f on P: the previous behavior is composed with f;

- run is used to send some data D to P: P returns its behavior applied to D;

- stop is used to stop P.

For security reasons, the process must only work with messages coming from its creator: other messages must be discarded.

Solutions

Es 1

```
(define (f . L)
  (foldl (lambda (x y)
           (list x y))
         (foldl cons ‘() L)
         L))
```

Es 2

```
data Listree a = Nil | Cons a (Listree a) | Branch (Listree a)(Listree a) deriving (Eq, Show)

exfig = Branch (Cons 1 Nil) (Cons 2 (Branch (Branch (Cons 3 Nil) (Cons 4 Nil))
                                     (Cons 5 (Cons 6 (Cons 7 Nil)))))

instance Functor Listree where
  fmap f Nil = Nil
  fmap f (Cons x y) = Cons (f x) (fmap f y)
  fmap f (Branch x y) = Branch (fmap f x) (fmap f y)

instance Foldable Listree where
  foldr f i Nil = i
  foldr f i (Cons x y) = f x (foldr f i y)
  foldr f i (Branch x y) = foldr f (foldr f i x) y

x <++> Nil = x
Nil <++> x = x
(Cons x y) <++> z = (Cons x (y <++> z))
(Branch x y) <++> z = (Branch x (y <++> z))

ltconcat t = foldr (<++>) Nil t
ltconcmap f t = ltconcat $ fmap f t

instance Applicative Listree where
  pure x = (Cons x Nil)
  x <*> y = ltconcmap (\f -> fmap f y) x
```

Es 3

```
cam(Beh, Who) ->
    receive
        {run, Who, What} ->
            Who ! Beh(What),
            cam(Beh, Who);
        {load, Who, Code} ->
            cam(fun (X) -> Code(Beh(X)) end, Who);
        {stop, Who} ->
            ok;
         _ -> cam(Beh, Who)
    end.
```