Principles of Programming Languages, 2016.07.22

FAMILY NAME ________________________

GIVEN NAME ________________________

DID YOU PRESENT A SMALL PROJECT? YES [] NO []

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 (9 pts)

1) Define the *iterate* function, with two parameters *f* and *v*, that returns the infinite list *(v f(v)* $f^2(v)$ *...* $f^n(v)$ *...)*.
Hint: use *delay* and *force*, as seen in class.
2) Define *take*, like in Haskell, to get items out of an infinite list.
E.g. *(take 10 (iterate (lambda (x) (+ x 1)) 0))* should return *(0 1 2 3 4 5 6 7 8 9)*

Exercise 2, Haskell (16 pts)

1) Define *iter* which works like *iterate* in the previous exercise. Note that Haskell already has *iterate*, but of course you cannot use it to define *iter*.
2) Consider this data type: *data Rf a b = Rf [a] (a -> b)*. Its first component is a list of values representing the *domain* of the second argument (a function). This means that the values returned from the function are meaningful only if its parameter is taken from the first list.
Is it possible to derive Show? Why? If the answer is no, make Rf an instance of Show, so that e.g. Rf [1,2,3] (+1) is represented as: "[1,2,3]-->[2,3,4]" (notice that [2,3,4] is the *image* of (+1) on the given domain).
3) Make (*Rf a)* an instance of Functor.
4) The *Rf* data type is used to represent functions. Given two *Rf* input values, say of type *(Rf a b)* and *(Rf b c)*, define a way to compose functions, i.e. write a function *compose* which returns a value of type *(Rf a c)*. Write the type of *compose*.
E.g. *compose (Rf [1,2,3] (+2)) (Rf [2,3,5] (*2))* should be *Rf [1,3] (\x -> (x+2)*2)*.

Exercise 3, Prolog (8 pts)

Define a predicate called sumoftwo, with takes an arbitrary nested list of numbers t, and a number n, and checks if it is possible to find two numbers x, y in t (that can be the same number), such that n = x+y. The predicate must be defined so that it is possible to use it to reach all the solutions.

E.g. sumoftwo([-1,2,[3],[4,[5]]], 7, X, Y).
X = 2,
Y = 5 ;
X = 3,
Y = 4

Solutions

Scheme

```
(define (iterate f v)
  (delay (cons v (iterate f (f v)))))

(define (take n prom)
  (if (= n 0)
    '()
    (let ((v (force prom)))
      (cons (car v) (take (- n 1) (cdr v))))))
```

Haskell

1)

```
iter f x = x : iter f (f x)
```

2)

No: the second component of Rf is a function, so it has not a standard representation.

```
instance (Show a, Show b) => Show (Rf a b) where
    show (Rf d f) = show d ++ "-->" ++ show (map f d)
```

3)

```
instance Functor (Rf a) where
    fmap f (Rf d g) = Rf d (f . g)
```

4)

The main idea is that the composition (Rf d1 f) (Rf d2 g) should be such that its domain is made of all the elements of d1 such that their f image is in d2, while the function part is the composition of g and f. Hence:

```
compose :: Eq b => Rf a b -> Rf b c -> Rf a c
compose (Rf d1 f) (Rf d2 g) = Rf [x | x <- d1, elem (f x) d2] (g . f)
```

Prolog

```
deepmember(X, [X|Xs]) :- atomic(X).
deepmember(X, [Y|Ys]) :- deepmember(X, Y) ; deepmember(X, Ys).

sumoftwo(L, V, X, Y) :- deepmember(X, L), deepmember(Y, L), V is X+Y.
```