Notes:
- Total available time: 2h.
- You may use any written material you need.
- You cannot use computers, phones or laptops during the exam.

Exercise 1, Scheme (8 pts)

Define a procedure, called ftree, which takes two nested lists (with any possible nesting depth), one containing functions and one containing data, and applies the functions to the data provided at the same position of the function. When ftree is called with an empty first parameter, it works like the identity function.

E.g.

```scheme
(define f1 (lambda (x) (+ 1 x)))
(define f2 (lambda (x) (* 2 x)))
(define f3 (lambda (x) (- x 10)))
(define f4 (lambda (x) (string-append "<<" x ">>")))
(define t1 '(1 (2 3 4) (5 (6)) ("hi!" 8)))
(define o1 `(,f1 (,f1 ,f2 ,f1) (,f3 (,f1)) (,f4 ,f3)))
(define o2 `(,f1 () (,f3 (,f1)) (,f4 ,f3)))
```

```
(ftree o1 t1) must return (2 (3 6 5) (-5 (7)) (<<hi!>> -2))
(ftree o2 t1) must return (2 (2 3 4) (-5 (7)) (<<hi!>> -2))
```

Exercise 2, Haskell (18 pts)

1. Define a generic tree data structure, called Gtree, for trees having any number of children.
2. Make Gtree an instance of Functor.
3. Make Gtree an instance of Applicative, with <*> working like ftree in Exercise 1, but for the empty first parameter (i.e. the two arguments of <*> must necessarily have the same structure).
4. Is it possible to define a <*> operation which works exactly like ftree (of course, with the hypothesis of having homogeneous Gtrees)? If the answer is yes, implement it; if no, explain why.

Exercise 3, Prolog (5 pts)

Define a “deep reverse” predicate, that takes a possibly nested lists, with any nesting depth, and reverse it and all its sub-lists.

E.g.

```
deeprev([1,[2,3],[4,[5]]], X)
X = [[[[5],4],[3,2],1]].
```
Solutions

Scheme

(define (ftree treef tree)
  (cond
    ((null? treef) tree)
    ((list? treef) (cons (ftree (car treef)(car tree))
                         (ftree (cdr treef)(cdr tree))))
    (else ; should be atoms
     (treef tree))))

Haskell

1) data Gtree a = Leaf a | Node [Gtree a] deriving (Show, Eq)

2) instance Functor Gtree where
   fmap f (Node []) = Node []
   fmap f (Leaf v) = Leaf (f v)
   fmap f (Node (x:xs)) = Node ((fmap f x) : vs) where
                          Node vs = fmap f (Node xs)

3) ftree :: Gtree (a -> b) -> Gtree a -> Gtree b
   ftree (Node []) (Node []) = Node []
   ftree (Leaf f) (Leaf v) = Leaf (f v)
   ftree (Node (f : fs)) (Node (x : xs)) = Node (ftree f x) : vs where
                                         Node vs = ftree (Node fs) (Node xs)
   ftree o v = error "bad structure"

   instance Applicative Gtree where
      pure = Leaf
      (<*>) = ftree

4) No, because <*> has type Gtree (a -> b) -> f a -> f b. To have a behavior like Scheme's ftree, we need an identity for the "missing parts" of the first argument. But this means that <*> must have type Gtree (a -> a) -> f a -> f a, which is impossible.

Prolog

deeprev([],[]) :- !.
deeprev([X|Xs], R) :- !, deeprev(X,V), deeprev(Xs,Vs), append(Vs, [V], R).
deeprev(X,X).