Exam

1. Given the following typeclass definition:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)       :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger         :: Integer -> a
```

and given the following definition of type `MyNatural`:

```
data MyNatural = Empty | () :-: MyNatural
    deriving (Eq, Show)
infixr 5 :-:
```

so that e.g.:

```
twoM = () :-: () :-: Empty
threeM = () :-: () :-: () :-: Empty
-- (or: threeM = () :-: twoM)
```

consider the following functions:

```
f1 Empty y = y
f1 (() :-: x) y = () :-: (f1 x y)
f2 Empty y = Empty
f2 (() :-: x) y = f1 y (f2 x y)
f3 x Empty = x
f3 Empty x = error "foo"
f3 (() :-: x) (() :-: y) = f3 x y
```

and make the following definition complete:

```
instance Num MyNatural where
    ...
```

Define appropriate auxiliary functions, if necessary.

Please note that the following equation must be obeyed in order to make `abs` and `signum` correctly defined:

```
(abs x) * (signum x) == x
```

`signum` is either 1 (positive argument), 0 (zero) or -1 (negative argument) in general case. For natural numbers that we try to define here, it may obviously be only zero or one. The same note applies to `negate` function: it should yield error on non-zero values, like in `f3` above.

2. Consider the following two versions of similarity score computations. The difference is in the expression defining value for `simEntry i j` .

   (a) Which of the versions is much faster than the other?

   (b) Why?

   Answering (a) but not (b) does not give much credit. Wrong answer is worth less than "I don't know".

   VERSION 1:

```
similScore :: String -> String -> Int
similScore xs ys = simScore (length xs) (length ys)
  where
     simScore i j = simTable!!i!!j
     simTable = [[ simEntry i j | j<-[0..]] | i<-[0..] ]

     simEntry :: Int -> Int -> Int
     simEntry 0 0 = 0
     simEntry i 0 = (i * scoreSpace)
     simEntry 0 j = (scoreSpace * j)
     simEntry i j = maximum [((simScore (i-1) (j-1)) + (score x y)),
                             ((simScore (i-1) j) + (score x '-')),
                             ((simScore i (j-1)) + (score '-' y))]
                  where
                     x = xs!!(i-1)
                     y = ys!!(j-1)
```

   VERSION 2:

```
similScore :: String -> String -> Int
similScore xs ys = simScore (length xs) (length ys)
  where
     simScore i j = simTable!!i!!j
     simTable = [[ simEntry i j | j<-[0..]] | i<-[0..] ]

     simEntry :: Int -> Int -> Int
     simEntry 0 0 = 0
     simEntry i 0 = (i * scoreSpace)
     simEntry 0 j = (scoreSpace * j)
     simEntry i j = maximum [((simEntry (i-1) (j-1)) + (score x y)),
                             ((simEntry (i-1) j) + (score x '-')),
                             ((simEntry i (j-1)) + (score '-' y))]
                   where
                     x = xs!!(i-1)
                     y = ys!!(j-1)
```

3. The function `unfoldr` may be defined as follows:

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of
              Nothing -> []
              Just (a,b) -> a : unfoldr f b
```

With a suitable function `g` it is possible to implement the prelude function

```
iterate :: (a -> a) -> a -> [a]
```

as:

```
iterate = unfoldr . g
```

(a) Determine the type of function `g`.

(b) Define the function `g`.

4. The `Functor` class is defined as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

It is mandatory that all instances of `Functor` should obey:

```
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

Assume the following definition of lists as a functor instance:

```
instance Functor [] where
  fmap g []     = []
  fmap g (x:xs) = fmap g xs ++ [g x]
```

Is this a correct definition of a functor instance? Why or why not?

5. Explain the concept of a *spark* in Haskell. How does it relate to the following three functions

```
seq, pseq, par ::  a -> b -> b
```

Explain what they do.

6. Type derivation

   (a) Find the type of `(.).(.)`

   (b) Given that

   ```
   map2 :: (a -> b, c -> d) -> (a, c) -> (b, d)
   ```

   find the destination type `e` of the following function:

   ```
   rulesCompile :: [(String, [String])] -> e
   rulesCompile = (map . map2) (words . map toLower, map words)
   ```

   (c) Given that

   ```
   transformationApply :: Eq a => a -> ([a] -> [a]) -> [a] -> ([a], [a])
                                                        -> Maybe [a]
   orElse :: Maybe a -> Maybe a -> Maybe a
   ```

   find the type of

   ```
   foldr1 orElse (map (transformationApply wildcard f x) pats)
   ```

Good Luck!