EDAN40

2nd June 2017

14:00 - 19:00

WRITE ONLY ON ONE SIDE OF THE PAPER - the exams will be scanned in and only the front/odd pages will be read.

DO NOT WRITE WITH OTHER COLOUR THAN BLACK - coloured text may disappear during scanning

PUT YOUR ID AND PAGE NUMBER ON EACH PAGE YOU SUBMIT - make sure that the amount of pages is equal to the amount you note on the front information page

WRITE CLEARLY - if we cannot read you we cannot grade you.

PRELIMINARY AMOUNT OF POINTS : 6 (one per question)

```
{-A list of selected functions from the Haskell modules:
  Prelude
  Data.List
  Data.Maybe
  Data.Char -}
------------------------------------------------------
-- standard type classes
class Show a where
  show :: a -> String

class Eq a where
  (==), (/=)           :: a -> a -> Bool

class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

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

class (Num a, Ord a) => Real a where
  toRational           :: a -> Rational

class (Real a, Enum a) => Integral a where
  quot, rem            :: a -> a -> a
  div, mod             :: a -> a -> a
  toInteger            :: a -> Integer

class (Num a) => Fractional a where
  (/)                  :: a -> a -> a
  fromRational         :: Rational -> a

class (Fractional a) => Floating a where
  exp, log, sqrt       :: a -> a
  sin, cos, tan        :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round      :: (Integral b) => a -> b
  ceiling, floor       :: (Integral b) => a -> b

-----------------------------------------------------
-- numerical functions

even, odd   :: (Integral a) => a -> Bool
even n      = n `rem` 2 == 0
odd         = not . even
-------------------------------------------------------
-- monadic functions

sequence     :: Monad m => [m a] -> m [a]
sequence     = foldr mcons (return [])
                 where mcons p q = do x <- p; xs <- q; return (x:xs)

sequence_    :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs; return ()

------------------------------------------------------
-- functions on functions

id          :: a -> a
id x        = x

const       :: a -> b -> a
const x _   = x

(.)         :: (b -> c) -> (a -> b) -> a -> c
f . g       = \x -> f (g x)

flip        :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

($)         :: (a -> b) -> a -> b
f $ x       = f x

-----------------------------------------------------
-- functions on Bools

data Bool = False | True

(&&), (||)  :: Bool -> Bool -> Bool
True  && x  = x
False && _  = False
True  || _  = True
False || x  = x
not          :: Bool -> Bool
not True    = False
not False   = True

------------------------------------------------------
-- functions on Maybe
data Maybe a = Nothing | Just a

isJust               :: Maybe a -> Bool
isJust (Just a)      = True
isJust Nothing       = False

isNothing            :: Maybe a -> Bool
isNothing            = not . isJust

fromJust             :: Maybe a -> a
fromJust (Just a)    = a

maybeToList          :: Maybe a -> [a]
maybeToList Nothing  = []
maybeToList (Just a) = [a]
```

```
listToMaybe          :: [a] -> Maybe a
listToMaybe []       = Nothing
listToMaybe (a:_)    = Just a

-------------------------------------------------------
-- a hidden goodie

instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)

--------------------------------------------------------
-- functions on pairs

fst         :: (a, b) -> a
fst (x, y)  = x

snd         :: (a, b) -> b
snd (x, y)  = y

curry       :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry     :: (a -> b -> c) -> (a, b) -> c
uncurry f p = f (fst p) (snd p)
--------------------------------------------------------
-- functions on lists

map      :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(++)             :: [a] -> [a] -> [a]
xs ++ ys         = foldr (:) ys xs

filter           :: (a -> Bool) -> [a] -> [a]
filter p xs      = [ x | x <- xs, p x ]

concat           :: [[a]] -> [a]
concat xss       = foldr (++) [] xss

concatMap        :: (a -> [b]) -> [a] -> [b]
concatMap f      = concat . map f

head, last       :: [a] -> a
head (x:_)       = x

last [x]         = x
last (_:xs)      = last xs

tail, init       :: [a] -> [a]
tail (_:xs)      = xs

init [x]         = []
init (x:xs)      = x : init xs
```

```
null             :: [a] -> Bool
null []          = True
null (_:_)       = False

length           :: [a] -> Int
length []        = 0
length (_:l)     = 1 + length l

(!!)             :: [a] -> Int -> a
(x:_)  !! 0      = x
(_:xs) !! n      = xs !! (n-1)

foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

iterate          :: (a -> a) -> a -> [a]
iterate f x      = x : iterate f (f x)

repeat           :: a -> [a]
repeat x             = xs where xs = x:xs

replicate            :: Int -> a -> [a]
replicate n x        = take n (repeat x)

cycle                :: [a] -> [a]
cycle []             = error "Prelude.cycle: empty list"
cycle xs = xs' where xs' = xs++xs'

take, drop           :: Int -> [a] -> [a]
take n _ | n <= 0    = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs

drop n xs | n <= 0   = xs
drop _ []            = []
drop n (_:xs)        = drop (n-1) xs

splitAt              :: Int -> [a] -> ([a],[a])
splitAt n xs         = (take n xs, drop n xs)

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []       = []
takeWhile p (x:xs)
      | p x          = x : takeWhile p xs
      | otherwise    = []

dropWhile p []       = []
dropWhile p xs@(x:xs')
      | p x          = dropWhile p xs'
      | otherwise    = xs
```

```
lines, words         :: String -> [String]
-- lines "apa\nbepa\ncepa\n" == ["apa","bepa","cepa"]
-- words "apa bepa\n cepa"   == ["apa","bepa","cepa"]

unlines, unwords     :: [String] -> String
-- unlines ["apa","bepa","cepa"] == "apa\nbepa\ncepa"
-- unwords ["apa","bepa","cepa"] == "apa bepa cepa"

and, or              :: [Bool] -> Bool
and                  = foldr (&&) True
or                   = foldr (||) False

any, all             :: (a -> Bool) -> [a] -> Bool
any p            = or . map p
all p            = and . map p

elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           = any (== x)
notElem x        = all (/= x)

lookup           :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []    = Nothing
lookup key ((x,y):xys)
    | key == x   = Just y
    | otherwise  = lookup key xys

sum, product     :: (Num a) => [a] -> a
sum              = foldl (+) 0
product          = foldl (*) 1

maximum, minimum :: (Ord a) => [a] -> a
maximum []       = error "Prelude.maximum: empty list"
maximum xs       = foldl1 max xs

minimum []       = error "Prelude.minimum: empty list"
minimum xs       = foldl1 min xs

zip              :: [a] -> [b] -> [(a,b)]
zip              = zipWith (,)

zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 = z a b : zipWith z as bs
zipWith _ _ _    = []

unzip            :: [(a,b)] -> ([a],[b])
unzip            = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

nub              :: (Eq a) => [a] -> [a]
nub []           = []
nub (x:xs)       = x : nub [ y | y <- xs, x /= y ]

delete           :: Eq a => a -> [a] -> [a]
delete y []      = []
delete y (x:xs)  = if x == y then xs else x : delete y xs

(\\)             :: Eq a => [a] -> [a]-> [a]
(\\)             = foldl (flip delete)

union            :: Eq a => [a] -> [a] -> [a]
union xs ys      = xs ++ ( ys \\ xs )

intersect                :: Eq a => [a] -> [a]-> [a]
intersect xs ys          = [ x | x <- xs, x `elem` ys ]

intersperse              :: a -> [a] -> [a]
-- intersperse 0 [1,2,3,4] == [1,0,2,0,3,0,4]

transpose                :: [[a]] -> [[a]]
-- transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]

partition                :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs           = (filter p xs, filter (not . p) xs)

group                    :: Eq a => [a] -> [[a]]
-- group "aapaabbbeee"   == ["aa","p","aa","bbb","eee"]

isPrefixOf, isSuffixOf   :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] _          = True
isPrefixOf _ []          = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys

isSuffixOf x y           = reverse x `isPrefixOf` reverse y

sort                     :: (Ord a) => [a] -> [a]
sort                     = foldr insert []

insert                   :: (Ord a) => a -> [a] -> [a]
insert x []              = [x]
insert x (y:xs)          = if x <= y then x:y:xs else y:insert x xs

------------------------------------------------------------
-- functions on Char

type String = [Char]

toUpper, toLower  :: Char -> Char
-- toUpper 'a'    == 'A'
-- toLower 'Z'    == 'z'

digitToInt        :: Char -> Int
-- digitToInt '8' == 8

intToDigit        :: Int -> Char
-- intToDigit 3   == '3'

ord               :: Char -> Int
chr               :: Int -> Char
```

Lund University
Department of Computer Science

EDAN40: Functional Programming
2nd June 2017, 14–19

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!