Examiner: David Sands, dave@chalmers.se, D&IT,
Planned visit to the examination halls approx 9 - 9.30 to answer questions. At other times by phone 0737207663

Functional Programming TDA 452, DIT 143

2022-01-11 08.30 – 12.30 Lindholmen

0737207663

- There are two parts to this exam. You must pass Part I to get at pass on the course. Part I has a total of 26 points; 15 points guarantees a pass on the exam (3/G).

  Part II has a total of 22 points and is graded if part I is passed. 9 points on part II guarantees a grade 4, 15 points on part II guarantees a grade 5/VG.

  The final grade boundaries may be slightly lower but will never be higher than those given. Any bonus points from 2021 will not be included in the reported grade; if you think your bonus points can make a difference to your reported grade (once you receive it) then please contact the examiner in slack and he will be happy to check this for you.
- Results: within approximately 10 days.
- **Permitted materials:**
  - Dictionary
- **Please read the following guidelines carefully:**
  - Read through all Questions before you start working on the answers.
  - Begin each whole question on a new sheet.
  - Write clearly; unreadable = wrong! **If you give multiple answers to the same question only the first answer will be graded.**
  - For each part Question, if your solution consists of more than a few lines of Haskell code, use your common sense to decide whether to include a short comment to explain your solution.
  - You can use any of the standard Haskell functions *listed at the back of this exam document*, as well as any functions from standard the standard type classes Monad, Functor, or Applicative.
  - **Full points** are given to solutions which are **short**, **elegant**, and **correct**. Fewer points may be given to solutions which are unnecessarily complicated or unstructured.
  - You are **encouraged to use** the solution to an **earlier part** of a Question to help solve a **later part** — even if you did not succeed in solving the earlier part.

Part I

1. *(4 points)* In this question you are to define a function

   ```
   range :: (Ord a,Num a) => [a] -> a
   ```

   which computes the difference between the largest and the smallest values in a given list of numbers. Your definition should satisfy the following property:

   ```
   prop_range as = not (null as) ==> range as == maximum as - minimum as
   ```

   Your definition should not use `minimum` or `maximum` (even if you implement them yourselves). Your definition should compute the range with a single traversal of the input list, by defining a single recursive helper function. Only use standard functions `(-)`, `max` and `min`. Hint: your helper function will need two additional parameters.

2. *(6 points)* For each of the following definitions, give the most general type, or write "No type" if the definition is not correct in Haskell.

   ```
   fa (x:y)      = (x,x,y)
   fb x y z      = x<=y && not z
   fc a          = do s <- a
                       putStrLn $ "Answer: " ++ s
   ```

3. *(6 points)* Given the following Data type

   ```
   data T = A Int | B Int T | C T T | D String
     deriving Show
   ```

   - *(2 points)* Define a value of type `T` which contains exactly one of each constructor of `T`.
   - *(4 points)* Define a function

     ```
     mapT :: (Int -> Int) -> T -> T
     ```

     so that, for example, if `t :: T` then `mapT (*2) t` produces a tree like `t`, except that every number in the tree is doubled.

4. *(6 points)* Consider the following types used to define the state of a game of tic-tac-toe (otherwise known as "noughts and crosses"):

```
type TicTac = [[Cell]]

type Cell   = Maybe XO

data XO = X | O deriving (Eq,Show)
```

Define a function

```
printTicTac :: TicTac -> IO()
```

which which displays a tictac on the screen with dashes between each row and vertical bar characters between each cell. For example, an empty three by three tictac with Xs on the middle row should be displayed as:

```
 | |
-----
X|X|X
-----
 | |
```

You may assume that the TicTac is a square grid (n lists of n cells, for some $n > 0$). For full marks your solution should make appropriate use of standard functions whenever possible, cleanly separate IO from pure computation, and, as always, use good Haskell style. Hint: the function `intersperse` (see function list) should be very useful here.

5. *(4 points)* Write a QuickCheck generator

```
spam :: Gen String
```

which is a generator for a random email address such as: `"bob7@gmail.com"`, `"alice99@hotmail.com"`, or `"dave@gmail.com"`. You should assume two Haskell definitions

```
names, emailProviders :: [String]
```

For example, we might have

```
names          = ["alice", "bob", "dave"]
emailProviders = ["gmail","yahoo","hotmail"]
```

Note that these are just lists, not generators. Using these definitions together with the standard QuickCheck functions (listed in the appendix) you should generate email addresses like the examples above, combining names, email providers (all assumed to be .com) with **zero, one, or two digits** after the name (so it should be possible to generate all of the names in the examples above).

Part II

1. *(4 points)* The sorting function `sortOn` from `Data.List` (but not given in the list of standard functions) has the following type:

```
sortOn :: Ord b => (a -> b) -> [a] -> [a]
```

   The intended behaviour can be deduced from the type, but here is an example:

```
prop_sortOn =
  sortOn snd [("Aardvark",2), ("Zebra",1)] == [("Zebra",1), ("Aardvark",2)]
```

   Give a definition of `sortOn`.

2. *(6 points)* Define a function

```
vectorOfUnique :: Eq a => Int -> Gen a -> Gen [a]
```

   which is similar to `vectorOf` but produces a list of non repeated values. More specifically, `vectorOfUnique n gen` should be a generator for a list of n unique values. You may assume that the supplied generator argument `gen` can generate sufficiently many distinct values.

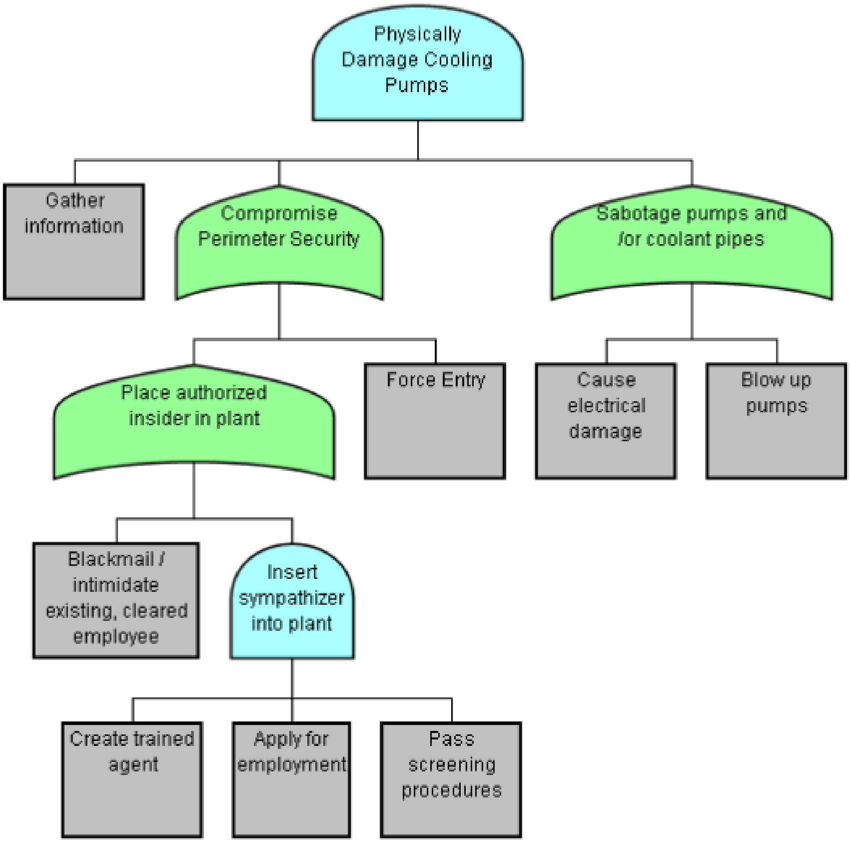
3. *(6 points)* Consider the following data types that model a small expression language with integer and boolean literals, and some arithmetic operations. In addition, the expression language also supports some comparison operators and an 'if-then-else' expression, which takes a condition (which is also an expression) and two subexpressions that model the 'then' and 'else' branch respectively. It also contains variables (which are labelled with their type).

```
data Expr = Num Int
          | Truth Bool
          | Var Type String
          | Arith ArithOp Expr Expr       -- Arithmetic operations
          | Compare CompareOp Expr Expr   -- Comparison operator
          | IF Expr Expr Expr             -- if-then-else
    deriving (Show)

data ArithOp = Mul | Add | Subtract      deriving Show

data CompareOp = Equal | Less | Greater  deriving Show

data Type = NumType | TruthType          deriving (Eq,Show)
```

   In this representation of expressions, not all expressions are valid. For example, performing an arithmetic operation on a Boolean, or comparing a Boolean with a number. Define a function `typeCheck :: Expr -> Maybe Type` which returns the type of a given expression (if it is well-typed). In this language boolean values can be compared as well as integers (just like in Haskell). Your function should determine the type without performing any evaluation, again, just like Haskell type checking. In particular, this means that in an `IF` expression, the first argument should represent a Boolean, and both branches (the second and third parameters) must have the same type.

4. *(6 points)* In security, attack trees are "hierarchical, graphical diagrams that show how low level hostile activities interact and combine to achieve an adversary's objectives - usually with negative consequences for the victim of the attack." (source: amenaza.com)

   Here is an example of an attack tree:



   In such a tree the nodes are attack goals and are of two kinds: *and-nodes*, like the root node of the example, and *or-nodes* such as the one labelled "Compromise Perimiter Security".

   (i) Give a data type to model attack trees so that trees such as the one above can be modelled. You should assume that nodes can have any number of branches (although in practice there are usually at least two, and sometimes many).

   (ii) Define a function which returns the highest number of branches that any single or-node has in a given attack tree. For the example above, the answer is two. If there are no or-nodes the answer should be zero.

```haskell
{- Some standard functions from Prelude Data.List
   Data.Maybe Data.Char Control.Monad -}
---------------------------------------------
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, 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,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, sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round, 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 = 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 Bool ------------------------
(&&), (||)        :: 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 -------------------------
isJust, isNothing      :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False
isNothing              =  not . isJust

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

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

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

catMaybes              :: [Maybe a] -> [a]
catMaybes ls           = [x | Just x <- ls]

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

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

swap              :: (a,b) -> (b,a)
swap (a,b)        = (b,a)

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            = foldr (const (1+)) 0

(!!)              :: [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 "cycle: empty list"
cycle xs          =  xs' where xs' = xs ++ xs'

tails             :: [a] -> [[a]]
tails xs          =  xs : case xs of
                          []      -> []
                          _ : xs' -> tails 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 (x:xs) | p x      = dropWhile p xs
                   | otherwise = x:xs

span :: (a -> Bool) -> [a] -> ([a], [a])
span p as = (takeWhile p as, dropWhile p as)

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 ["ap","bep","cep"] == "ap\nbep\ncep"
-- unwords ["ap","bep","cep"] == "ap bep cep"

reverse          :: [a] -> [a]
reverse          =  foldl (flip (:)) []

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 (x:xs) = foldl max x xs

minimum [] = error "Prelude.minimum: empty list"
minimum (x:xs) = foldl min x 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 = groupBy (==)

groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []     =  []
groupBy eq (x:xs) =  (x:ys) : groupBy eq zs
              where (ys,zs) = span (eq x) xs

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

isSuffixOf :: Eq a => [a] -> [a] -> Bool
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]

isSpace, isDigit :: Char -> Bool
toUpper, toLower :: Char -> Char

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

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

ord :: Char -> Int
chr :: Int  -> Char
--------------------------------------------
-- Useful functions from Test.QuickCheck
arbitrary :: Arbitrary a => Gen a
-- generator used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- a random element in the given inclusive range.

oneof :: [Gen a] -> Gen a
-- Randomly uses one of the given generators
frequency :: [(Int, Gen a)] -> Gen a
-- Chooses from weighted list of generators

elements :: [a] -> Gen a
-- Generates one of the given values.

listOf :: Gen a -> Gen [a]
-- Generates a list of random length.
vectorOf :: Int -> Gen a -> Gen [a]
-- Generates a list of the given length.

sized :: (Int -> Gen a) -> Gen a
-- construct generators that depend a size param.
```