# Exam – Introduction to Functional Programming

TDA555/DIT440, HT-21
Chalmers och Göteborgs Universitet, CSE

*Day:* 2021-10-27, *Time:* 14:00-18:00, *Place:* Johanneberg

**Course responsible**

Alex Gerdes (031-772 6154). He will visit the exam room once between 15:00 and 15:30, and after that is available by phone.

**Allowed aids**

An English dictionary.

**Grading**

The exam consist of two parts. Successfully completing Part I gives a 3 or G; Part I and Part II are both needed for a 4, 5, or VG.

**Part I** This part has seven small assignments. Give good enough answers for 5 assignments here and you will get a 3 or G.

**Part II** This part has 2 more advanced questions.

- You do not need to solve this part if you are happy with a 3 or G!
- An answer with minor mistakes might be accepted, but this is at the discretion of the marker. An answer with large mistakes will be marked as incorrect.
- Pass Part I and one assignment of your choice here and you will get a 4.
- Pass Part I and both assignments here and you will get a 5 or a VG.

**Notes**

- Begin each assignment on a new sheet.
- Write your number on each sheet.
- You may write your answers in Swedish and English.
- Excessively complicated answers might be rejected.
- *Write legibly!* Solutions that are difficult to read are marked as incorrect!
- You can make use of the standard Haskell functions and types given in the attached list (you have to implement other functions yourself if you want to use them).
- You do not have to import standard modules in your solutions. You do not have to copy any of the code provided.
- **Good luck!**

# Part I

You have to complete 5 out of the following 7 assignments to get a pass on the exam.

## 1

Given the following definitions:

```
thrice x = [x, x, x]

sums (x:y:ys) = x : sums (x + y : ys)
sums xs       = xs
```

What does the expression `map thrice (sums [0 .. 4])` evaluate to? Write down the intermediate steps of your computation. If the type of your answer is incorrect then your solution will be considered incorrect.

In addition, what are the (most general) types of `thrice` and `sums`?

## 2

Write a function `nRoots`:

```
nRoots :: Double -> Double -> Double -> Int
```

that, for some $a$, $b$, and $c$, determines the number of solutions (real roots) of a quadratic equation:

$$ax^2 + bx + c = 0$$

using *guards*.

Use this `nRoots` function to define a function that calculates the real roots of such an equation with the following type signature:

```
data Root = None | One Double | Two Double Double

roots :: (Double, Double, Double) -> Root
```

using *pattern matching* (case distinction). The `roots` function uses the data type `Root` to return the right number of solutions.

*Hint*: the roots of a quadratic equation are: $\frac{-b+\sqrt{D}}{2a}$ and $\frac{-b-\sqrt{D}}{2a}$, $\frac{-b}{2a}$, or none if the discriminant ($b^2 - 4ac$) is positive, zero, or negative respectively.

## 3

Your task is to write a function that conditionally maps a given function over a list:

```
condMap p f xs = ...
```

The `condMap` function takes a predicate `p` (a function returning a boolean) and function `f` and applies this function to each element in the list (`xs`) if and only if the predicate `p` holds. Start by giving the type signature for `condMap`. Next, implement this function three times using the following techniques:

1. using explicit recursion
2. using a list comprehension
3. using the higher-order function `map`

Use the `condMap` function to define the following function:

```
replace x y xs = ...
```

that replaces each element in `xs` that is equal to `x` with `y`. For example:

```
ghci> replace 'e' 'u' "Alex Gerdes"
"Alux Gurdus"
```

In addition, give the most general type signature for `replace`.

## 4

Recall the `Root` data type defined in Assignment 2:

```
data Root = None | One Double | Two Double Double
```

Write an instance declaration for the type class `Show` for the `Root` data type. In case there are no roots the `show` function should return `"no roots"`, if there is a single root the string representation should display the value of the real root number, and finally in case of two roots, the `show` function should return a string with the two roots separated by a space. For example:

```
ghci> show None
"no roots"
ghci> show (One 3.0)
"3.0"
ghci> show (Two 2.0 3.0)
"2.0 3.0"
```

We continue with the `Eq` type class. Give an instance declaration for the `Eq` type class for the `Root` data type. Two values of the `Root` type are equal to each other if and only if:

- both are `None`,
- both are a single root and the associated `Double` values are equal,
- both have two roots that are the same, which means that the associated `Double` values are equal. Note that the order of the roots in a `Two` constructor is irrelevant, that is, `Two 2.0 3.0` is equal to `Two 3.0 2.0`.

## 5

Write a function that takes the three coefficients of a quadratic equation as argument:

```
askRoots :: (Double, Double, Double) -> IO ()
askRoots (a, b, c) = ...
```

and asks a user to give the roots of the corresponding quadratic equation $ax^2 + bx + c$. If a user answers correctly the function should display an appropriate message; if not, then the function should show what the user should have answered. The following excerpt shows an example interaction:

```
ghci> askRoots (2, 5, 3)
What are the roots of: 2.0x^2 + 5.0x + 3.0? Write '-' for none.
> 3.0
Bummer, the roots are: -4.0 -6.0

ghci> askRoots (2, 5, 3)
What are the roots of: 2.0x^2 + 5.0x + 3.0? Write '-' for none.
> -6.0 -4.0
Well done!
```

You may reuse the functions from Assignment 2 and 4, even if you have not implemented them. Just assume that the function `roots`, and the instance declarations do what they are supposed to do; you can do this assignment independent from the other assignments.

*Hint*: start with defining the following help function:

```
readRoots :: String -> Root
```

that takes a string containing either a single dash (`"-"`), a single real number, or two real numbers (separated by a space) and returns a value of type `Root`. The single dash should be translated to `None`, and the single and double real numbers to `Root` values using the `One` and `Two` constructors respectively.

## 6

The function `sort` from the `Data.List` library sorts an input list in increasing order. We want to test the implementation of this function with QuickCheck. Your tasks are:

1. Write a property that checks that the `sort` function returns a list of the same length as the input list.
2. Write a property that checks that the first element in the result list is the smallest element in the input list.
3. Write a property that validates that the elements in the result list are indeed in increasing order.

## 7

A shop sells three kinds of skis: skate skis, classic skis, and skin skis of different brands and having a particular price. In most cases, ski sizes are standardized into six different sizes: from 182 to 207 cm, with an interval of 5 cm. However, it is also possible for skis to have a custom (integral) size (the size of the ski in centimeters). The skis also have a binding, and in case of classic or skin skis we use either a normal or a moveable binding. The moveable bindings come in two variations: race and switch. Skate skis have a skate binding, which should only be used on skate skis.

Your task are:

1. Define data types that model the above situation as precisely as possible.
2. Write a function that retrieves all the classic skis from the shop.

# Part II

You do not need to work on this part if you only want to get a 3 or a G.

## 8

Consider the function `permutations` from the `Data.List` library, which computes all the possible permutations of elements in a list. For example:

```
ghci> permutations [1,2,3]
[[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
```

Your task is to write QuickCheck properties to verify this function:

1. Write a property that checks that the correct number of permutations is generated.
2. Write a function `isPerm :: Eq a => [a] -> [a] -> Bool` that verifies that the two argument lists are permutations of each other.
3. Write a property that validates that every list in the output of `permutations` is a permutation of the input.

The number of permutations grows *very rapidly* and you should only test the properties with random lists of limited size. Write a new auxiliary (help) data type for small lists, and make this data type an instance of the `Arbitrary` type class, which makes sure to only generate random lists with a maximum length of 8. Describe how you must change the properties that you have defined earlier to use your small list data type.

## 9

Consider the following data types that model a small expression language with integer and boolean literals, addition, and (integer) division. In addition, the expression language also supports a comparison operator (greater than) 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.

```haskell
data Expr
  = Lit Literal          -- Literal, either integer or boolean
  | Add Expr Expr        -- Addition
  | Div Expr Expr        -- Integer division
  | Gth Expr Expr        -- Comparison operator
  | If  Expr Expr Expr   -- if-then-else
  deriving (Show)

data Literal
  = N Int                -- Integer literal
  | B Bool               -- Boolean literal
  deriving (Eq, Show)
```

Using these data types we can define some example expressions:

```
-- Smart constructor for integers
num :: Int -> Expr
num = Lit . N

-- Smart constructor for booleans
bool :: Bool -> Expr
bool = Lit . B

-- Example expressions
e1, e2 :: Expr
e1 = (num 4 `Add` num 5) `Div` num 2  -- (4 + 5) `div` 2
e2 = If (e1 `Gth` (num 4))            -- if e1 > 4 then 3 else 4 + 1
       (num 3)
       (num 4 `Add` num 1)
```

The above examples are well-formed, which means that we can evaluate these expressions and get a result (an integer or a boolean). The expression language model also allows for ill-formed expressions, for example:

```
e3_bad, e4_bad :: Expr
e3_bad = If (num 4) (bool True) (num 0)               -- non-boolean condition
e4_bad = If (bool False) (num 4) (num 3 `Div` num 0)  -- division by zero
```

The expression `e3_bad` is ill-formed because the condition is of an integer type, whereas it should be an expression that evaluates to a boolean. The expression `e4_bad` contains a division by zero in the 'else' branch.

Your task is to write an evaluation function with the following type:

```
eval :: Expr -> Maybe Lit
```

This function evaluates the expression and returns the result in a `Maybe`. If the expression is ill-formed or contains a division by zero (that needs to evaluated), the `eval` function should return `Nothing`, indicating that something has gone wrong. For example, `eval e2` should evaluate to `Just (N 3)`, and `eval e3_bad` to `Nothing`.

Note that you don't need to handle the comparison of boolean literals, in such case you can return `Nothing`.

```haskell
{- This is a list of selected functions from the
   standard Haskell modules: Prelude Data.List
   Data.Maybe Data.Char Control.Monad         -}
-----------------------------------------
-- * Standard type classes

class Show a where  show :: a -> String

class Read a where  read :: String -> a

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 ()

liftM   :: Monad m => (a -> b) -> m a -> m b
liftM f m1   = do x1 <- m1
                  return (f x1)
-----------------------------------------
-- * 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,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]

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 "Prelude.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 xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  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 ["apa","bepa","cepa"]
--       == "apa\nbepa\ncepa\n"
-- unwords ["apa","bepa","cepa"]
--       == "apa bepa cepa"

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]

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
-----------------------------------------
-- * Useful functions from Test.QuickCheck

arbitrary :: Arbitrary a => Gen a
-- the generator for values of a type
-- in class Arbitrary, used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- Generates 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 list of generators with
-- weighted random distribution.

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 on
-- the size parameter.
-----------------------------------------
-- * Useful IO function
putStr, putStrLn  :: String -> IO ()
getLine           :: IO String

type FilePath = String
readFile     :: FilePath -> IO String
writeFile    :: FilePath -> String -> IO ()
```