# EDAN40

## examination

# 4 hp

12th April 2023

8:00 - 13:00

**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 OR DARK BLUE** - lightly 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 (properly) grade you.

**PRELIMINARY MAX AMOUNT OF POINTS: 6**

Lund University
Department of Computer Science

EDAN40: Functional Programming
12th April 2023, 8–13

# Exam

1. **Point-free notation**

   Rewrite the following two definitions into a point-free form (i.e., `f = ...`, `g = ...`), using neither lambda-expressions nor list comprehensions nor enumeration nor `where` clause nor `let` clause:

   ```
   f x y = (42 - y) * x
   g x y = y x
   ```

2. **Type derivation**

   Find the types of the following expressions:

   ```
   (($) $)
   ((.) .)
   ((:) :)
   ((==) ==)
   ((||) ||)
   ```

3. **Proving program properties**

   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 `Maybe` types as a functor instance:

   ```
   instance Functor Maybe where
       fmap f (Just x) = Just (f x)
       fmap f Nothing = Nothing
   ```

   Is this a correct definition of a functor instance? Why or why not? **Prove your claim.**

4. **Evaluation**

   Explain what a *thunk* is.

5. **Monadic computations**

   What is the type of `e` defined below? Motivate your answer.

```
e k = do
     x <- k
     Nothing
     return False
```

6. **Types and type classes**

   - Define a tree data structure so that the trees are ternary (i.e., each node has either three children or is a leaf) and store strings in leaves.
   - Generalize your definition so that your ternary trees can contain objects of an arbitrary predetermined type in a leaf.
   - Assuming your polymorphic trees type is denoted by `Tree3 a` (you may use your own name used above), write all necessary code so that the following function is correct:

     ```
     myLength :: Tree3 String -> Tree3 Integer
     myLength = fmap length
     ```

     and yelds a ternary tree with leaves containing lengths of the strings placed in the respective leaves of the argument tree.
     Make sure that the following works as well:

     ```
     myReverse :: Tree3 String -> Tree3 String
     myReverse = fmap reverse
     ```

# Good Luck!

```haskell
{-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]
```

```haskell
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
```

```haskell
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
```

230412

N40-0001-DTN

Ableton



1. f = flip \$ (*) . (42 -)

g = flip (\$)

0.9

2\. a) $(1.\$) :: (a \to b) \to a \to b$
$(2.\$) :: (c \to d) \to c \to d$ } renaming to distinguish between them

$a := c \to d$

$b := c \to d$

Substitute and remove first argument:

$(\$)\ \$ :: (c \to d) \to c \to d$

Beautify expression:

Answer: $((\$)\ \$) :: (a \to b) \to a \to b$

0.2

b) $((.2)\ .1)$ → renaming to distinguish between the functions

$(.1) :: (b \to c) \to (a \to b) \to a \to c$

$(.2) :: (e \to f) \to (d \to e) \to d \to f$

$b := e \to f$

$c := (d \to e) \to d \to f$

Substitute and remove first argument:

$((.)\ .) :: (a \to e \to f) \to a \to (d \to e) \to d \to f$

Beautify expression:

Answer: $((.)\ .) :: (a \to c \to d) \to a \to (b \to c) \to b \to d$

0.2

2. c) $((:1)\ :2)$ → Renaming to distinguish between them.

$(:1)$ : $a \to [a] \to [a]$

$(:2)$ : $b \to [b] \to [b]$

$a := b \to [b] \to [b]$

Substitute and remove first argument:

$((:)\ :) :: [(b \to [b] \to [b])] \to [(b \to [b] \to [b])]$

0.2

d) $((==1)\ ==2)$ → Renamins for distiction

$(==1) :: Eq\ a \Rightarrow a \to a \to Bool$

$(==2) :: Eq\ b \Rightarrow b \to b \to Bool$

$((==)\ ==)$ : Type error because $(b \to b \to Bool)$ is not of instance Eq.

D 0.2

e) $((||)\ ||) :: Bool \to Bool \to Bool \to Bool$

0

3. Law 1: $fmap\ id = id$

1. $fmap\ f\ (Just\ x) = Just\ (f\ x)$

$fmap\ id\ (Just\ x) = Just\ (id\ x) = Just(x) = id\ Just(x)$

2. $fmap\ f\ Nothing = Nothing$

$fmap\ id\ Nothing = Nothing = id\ Nothing$

Law 1 holds.

Law 2: $fmap\ (p\ .\ q) = (fmap\ p)\ .\ (fmap\ q)$

1. $fmap\ f\ (Just\ x) = Just\ (f\ x)$

$fmap\ (p\ .\ q)\ (Just\ x) = Just((p\ .\ q)\ x) = Just(p(q\ x))$

$((fmap\ p)\ .\ (fmap\ q))\ (Just\ x) = (fmap\ p)\ (fmap\ q\ (Just\ x)$

$= (fmap\ p)\ (Just\ (q\ x)) = Just\ (p\ (q\ x))$

2. $fmap\ f\ Nothing = Nothing$

$fmap\ (p\ .\ q)\ Nothing = Nothing$

$((fmap\ p)\ .\ (fmap\ q))\ Nothing = (fmap\ p)\ (fmap\ q\ Nothing$

$= fmap\ p\ Nothing = Nothing$

Law 2 holds.

1

Answer: Yes the definition is correct since both laws hold.

4. Thunk is related to the concept of lazy evaluation where expressions are not evaluated until being used.

0.2

5. e :: Maybe a → Maybe Bool

Because the expression in do has a Nothing, we can extract that the Monad being used is Maybe, which has values Nothing and Just. Because a monad always returns a monad and we can see that the function returns False, we can therefore conclude that the output type is Maybe Bool.

1

Ableton

6.

- data Tree3 String = Leaf String | Node ~~String~~ (Tree3 String) (Tree3 String) (Tree3 String)

- data Tree3 a = Leaf a | Node ~~a~~ (Tree3 a) (Tree3 a) (Tree3 a) deriving Show

- instance Functor (Tree3 a) where

```
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node ~~x~~ x y z) = Node ~~(f n)~~ (fmap f x)
       (fmap f y) (fmap f z)
```

0.9

# EDAN40

## examination

# 4 hp

2nd June 2022

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 OR DARK BLUE** - lightly 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 (properly) grade you.

**PRELIMINARY MAX AMOUNT OF POINTS: 6**

Lund University
Department of Computer Science

EDAN40: Functional Programming
25th April 2022, 14–19

# Exam

1. **Type derivation** (1p)

   (a) Assume that the type of reduce is

   ```
   reduce :: a -> a
   ```

   Find the type of

   ```
   prepare = reduce . words . map toLower . filter
                                             (not . flip
                                                 elem ".,:;*!#%&|")
   ```

   (b) Given that

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

   find the destination type b of the following function:

   ```
   rulesCompile :: [(String, [String])] -> b
   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)
   ```

2. **Proving program properties** (2p)

   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)
   ```

   Let Either be defined as follows:

   ```
   data Either a b = Left a | Right b
   ```

   Assume the following definition of Either types as a functor instance:

   ```
   instance Functor (Either a) where
       fmap f (Right x) = Right (f x)
       fmap f (Left x) = Left x
   ```

```
{-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]
```

```haskell
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
```

```haskell
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
```

230602

NH00039NFK
248LC66PY

Q1 (a). For simplicity, denote

- t1 u = elem u ".,:;*! #%&|"
- t2 = flip elem ".,:;*! #%&|"
- t3 = filter (not . t2)
- t4 = map toLower . t3

∵ elem :: (Eq a) ⇒ a → [a] → Bool.

∴ For t1 u, since type of a is Char, u has the type Char.

∵ flip :: (a → b → c) → b → a → c

flip elem :: (Eq a) ⇒ [a] → a → Bool

∴ t2 :: Char → Bool (Char is in class Eq)

∴ not . t2 :: Char → Bool

∴ t3 :: [Char] → [Char]

∵ toLower :: Char → Char

∴ t4 :: [Char] → [Char], or String → String.

∵ words :: String → [String]

∴ words . t4 :: String → [String]

∴ prepare :: String → [String].

Q1 (b). ∵ words . map toLower :: String → [String]

map words :: [String] → [[String]].

map . map2 :: (a → b, c → d) → [(a, c)] → [(b, d)]

∴ rulesCompile :: [(String, [String])] → [([String], [[String]])]

h :: [( [String] , [[String]] )]

Q1. (c). For simplicity, denote

$$t = \text{transformationApply wildcard } f \ x$$

(assume wildcard, f, x are fixed so that t does not depend on them)

Then $t :: \text{Eq } a \Rightarrow ([a], [a]) \to \text{Maybe } [a]$.

So, $\text{map } t \text{ pats} :: \text{Eq } a \Rightarrow [\text{Maybe } [a]]$

$\because$ $\text{foldr1} :: (a \to b \to b) \to [a] \to b$.

$\therefore$ $\text{foldr1 orElse } (\text{map } t \text{ pats}) :: \text{Eq } a \Rightarrow \text{Maybe } [a]$.

Q2. Proof 1: $\text{fmap id} = \text{id}$

Suppose $x = \text{Left } y$, then

$$\begin{aligned}\text{fmap id } x &= \text{fmap id } (\text{Left } y)\\ &= \text{Left } y\\ &= x\end{aligned}$$

Suppose $x = \text{Right } y$, then

$$\begin{aligned}\text{fmap id } x &= \text{fmap id } (\text{Right } y)\\ &= \text{Right } (\text{id } y)\\ &= \text{Right } y\\ &= x.\end{aligned}$$

Hence $\text{fmap id} = \text{id}$.

Proof 2: $\text{fmap } (p \cdot q) = (\text{fmap } p) \cdot (\text{fmap } q)$. We let $f' = p \cdot q$.

Suppose $x = \text{Left } y$, then

$$\text{fmap } f' \ (\text{Left } y) = \text{Left } y$$

Also, we have

$$\begin{aligned}&((\text{fmap } p) \cdot (\text{fmap } q))(\text{Left } y)\\ &= \text{fmap } p \ (\text{fmap } q \ (\text{Left } y))\\ &= \text{fmap } p \ (\text{Left } y)\\ &= \text{Left } y.\end{aligned}$$

Suppose $x = \text{Right } y$, then,

$$\text{fmap } f' \ (\text{Right } y) = \text{Right } (f' \ y)$$

Also, we have

$$\begin{aligned}&((\text{fmap } p) \cdot (\text{fmap } q)) \ (\text{Right } y)\\ &= \text{fmap } p \ (\text{fmap } q \ (\text{Right } y))\\ &= \text{fmap } p \ (\text{Right } (q \ y))\\ &= \text{Right } (p \ (q \ y))\\ &= \text{Right } ((p \cdot q) \ y) = \text{Right } (f' \ y)\end{aligned}$$

Q2. cont'd) Since the definition of the functor instance obeys the two laws suggested, the functor instance is correct. NH00039NFK 248LC66PY

Q3. - Spark is the potential of thread creation.

- It occurs in Haskell when users call 'par' function for parallel programming

- Spark is good for its lazy evaluation. Since it only evaluate the first argument in 'par' function to WHNF, the program does not require to evaluate the exact value of the first argument. This is helpful when exact value is hard to be evaluated, say, an infinite list results in the argument

Q4. (a).

paste reg im1 im2 = \pos -> if (reg pos) then (im1 pos) else (im2 pos)

~~| reg pos = im1 pos~~

~~| otherwise = im2 pos~~

Q4 (b). ~~Define inRange :: Region such that~~

~~inRange (x, y) = (inUnit x) && (inUnit y)~~

~~where inUnit z = (z >= 0) && (z <= 1)~~

lift0 val = const val

lift1 func im1 = func . im1

lift2 func im1 im2 = \p -> func (im1 p) (im2 p)

Q4(c). The operator (-) is in class Num, which is the child of Eq and Show classes. Hence before declaring the operator (-), we have to first declare

- show :: Image a -> String (in Show Class instance)
- (==) :: Image a -> Image a -> Bool (in Eq Class instance)

Then we can declare (-) :: Image a -> Image a -> Image a in Num Class instance. which may make use of lift2 function in Q4(b).

NH00039NFK
248LC66PY

Q5. Rewrite the function as

e k = k >>= \x → Nothing >>= \y → return 42

Recall (>>=) :: (Monad m) ⇒ m a → (a → m b) → m b

- Focus on Nothing >>= \y → return 42

∵ Nothing :: Maybe a

∴ return 42 :: (Num b) ⇒ Maybe b

So the type of this subfunction is (Num b) ⇒ Maybe b.

However since the first argument is Nothing, the function of the 2nd argument is not conducted, hence it returns Nothing.

Hence e k = k >>= \x → Nothing.

- Hence k :: Maybe a1

As a result, e :: (Num b) ⇒ Maybe a → Maybe b , and it returns Nothing for all k :: Maybe a.

# EDAN40

## examination

# 4 hp

18th August 2022

8:00 - 13:00

**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 OR DARK BLUE** - lightly 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 (properly) grade you.

**PRELIMINARY MAX AMOUNT OF POINTS: 6**

Lund University
Department of Computer Science

EDAN40: Functional Programming
18th August 2022, 8–13

# Exam

1. **Proving program properties (1.5p)**

   Given the following function:

```
foo :: [a] -> [a]
foo []     = []
foo (x:xs) = foo xs ++ [x]
```

   **prove** that the following holds:

```
foo (foo xs) = xs
```

2. **Type derivation (1p)**

   Find types of: a) `(:(.))`, b) `((:).)`, c) `(.(:))`, d) `((.):)`.

3. **Types and type classes (1.5p)**

   - (0.3p) Define a tree data structure so that the trees are ternary (i.e., each node has either three children or is a leaf) and store strings in each node.
   - (0.2p) Generalize your definition so that your ternary trees can contain objects of an arbitrary predetermined type in a node.
   - (1p) Assuming your polymorphic trees type is denoted by `Tree3 a`, write all necessary code so that the following function is correct:

```
myLength :: Tree3 String -> Tree3 Integer
myLength = fmap length
```

     and yelds a ternary tree with nodes containing lengths of the strings placed in the respective nodes of the argument tree.
     Make sure that the following works as well:

```
myReverse :: Tree3 String -> Tree3 String
myReverse = fmap reverse
```

4. **Do notation (1p)**

   Given the following function:

```
f x y = do
  a <- x
  b <- y
  return (a*b)
```

What is the type of `f`?
What is the value of `f [1,2,3] [2,4,8]` ?
What is the value of `f (Just 5) Nothing` ?
Is the expression `fmap (+2) (Just 5)` correct?
What is the type of expression `return 5`?

5. **Memoization (1p)**

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

   (a) (0.1p) Which of the versions is much faster than the other?
   (b) (0.9p) Why?

   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)
```

# Good Luck!

```haskell
{-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 [])
```

```haskell
                 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]
```

```haskell
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
```

```haskell
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
```

```haskell
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
```

220818

1

($1)

N40-0017-SHA

1.

```
foo :: [a] -> [a]
foo []     = []
foo (x:xs) = foo xs ++ [x]
```

Induction

Base case: xs = []

foo (foo xs) = foo (foo []) = foo [] = [] = xs

0.3

Hypothesis: Assume that for xs, foo (foo xs) = xs

Now we need to prove that if the hypothesis holds, that implies that the property holds for (x:xs)

assume:

Induction step:

foo (foo (x:xs)) = foo (foo xs ++ [x]) =

this step needs additional proof!

$= \text{foo } [x] \text{ ++ } \underbrace{\text{foo (foo xs)}}_{= \text{xs through hypothesis}} = \text{foo } [x] \text{ ++ xs} = ([\,] \text{ ++ } [x]) \text{ ++ xs} =$

= [x] ++ xs = (x:xs)

Because we have showed that if

foo (foo xs) = xs ⟹ foo (foo (x:xs))

and that the base case holds, we have shown that

0.6

foo (foo xs) = xs

2

($)
N90-0017-SHA

a) $(.) :: (b \to c) \to (a \to b) \to a \to c$

$(:) :: a \to [a] \to [a]$

Answer: The second argument of (:) must be a list, because (.) is not a list, (:(.)) will result in an error.

0.25

b)

Answer: $((:).) :: (a1 \to a2) \to a1 \to [a2] \to [a2]$

0.25

c)

Answer: $(.(:)) :: ([a] \to [a] \to c) \to a \to c$

0.25

d)

Answer: $((.):) :: [(b \to c) \to (a \to b) \to a \to c] \to$
$\to [(b \to c) \to (a \to b) \to a \to c]$

0.25

3. (3)

(8) N40-0027-SHA

0.3

a) data Tree = Leaf String | Branch String Tree Tree Tree

0.2

b) data Tree3 a = Leaf a | Branch a (Tree a) (Tree a) (Tree a)

c) instance Functor Tree3 where

fmap f (Leaf a) = Leaf (f a)

fmap f (Branch a x y z) = Branch (f a) (fmap f x) (fmap f y) (fmap f z)

Answer: When we make Tree3 a functor and implement fmap both myLength and myReverse will work as intended.

1

(4)

($)

N40-0017-SHA

9. 
```
f x y = do
    a <- x
    b <- y
    return (a*b)
```

a) f :: (Monad m, Num b) => m b -> m b -> m b

0.2

b) The value of f [1,2,3] [2,4,8] is:

[2,4,8,4,8,16,6,12,24]

0.2

c) The value of f (Just 5) Nothing is:

Nothing

0.2

d) The expression is correct, it evaluates to (Just 7)

0.2

e) (return 5) :: (Monad m, Num a) => m a

0.2

5. Memoization

(5)

(b)
N90-0017-SHA

a) The first version is much faster

b) The first version [illegible] the tabularized values from sumScore [illegible] makes use of previous computations to reduce the [illegible] time.

The second version [illegible] doesn't use previous [illegible] and instead just recursively tries to compute the value [illegible] be more computations and [illegible]