Functional Programming

Exam - Winter Term 2019/20

Prof. Dr. Sibylle Schupp

ANY WRITING ON THE EXAM SHEET, BE IT BEFORE THE START SIGNAL OR AFTER THE END SIGNAL, AUTOMATICALLY RESULTS IN A FAILED EXAM. THIS ALSO INCLUDES WRITING YOUR NAME AND MATRICULATION NUMBER AFTER THE END SIGNAL!

Last Name, First Name: ______________________
Matriculation Number: ______________________
Study Course: ______________________
Signature: ______________________

a) Place your student card and identity card clearly visible on your desk.

b) Complete the exam in *90 minutes*. Permitted aid: Dictionary. **No** other material is permitted.

c) Ensure that all electronic devices are turned **off**.

d) Use a **pen** to write your solutions (**red** color is **not** allowed). Pencil solutions will **not** be graded.

e) Only **one solution per question**. Multiple answers means no answer.

f) You may answer the questions in either **German or English**.

g) The exam consists of **eight** questions of varying weight. Points for each question is given in the margin. There are **100 points in total**.

h) The handout contains space for your solutions. Do not use your own sheets of paper. If you need additional space, ask the exam proctor.

i) If you obtain additional sheets from the exam proctor, you must enter your names and matriculation number on every sheet. Clearly indicate to what exercise the extra sheet belongs.

j) Inform the exam proctor when you need to leave the examination room. You are not allowed to leave the room without permission.

1	2	3	4	5	6	7	8

BP	Σ	Grade

1 Types and Type Classes

[7 points] a) What are the most general types (as returned in GHCi using `:type`) of the following expressions? Answer using Haskell type notation!

i) `True`

ii) `("False", not False)`

iii) `[take, drop]`

iv) `(tail [1,2,3], tail (show [1,2,3]))`

"Haskell type notation" means the way Haskell accepts types in contracts. For example the type of the expression `[True]` is denoted as `[Bool]`, not as "A list of Booleans."

[6 points] b) Consider the following function definitions and their partial contracts. The contracts are missing class constraints. For each function, tick off which class constraints are missing for the most general contract, and which are not.

<table>
<tr><td>Partial Contract</td><td colspan="2"><code>prntBtwnNth :: (?) => a -> [b] -> [b] -> [b]</code></td></tr>
<tr><td>Function</td><td colspan="2"><code>prntBtwnNth n del str =</code>
<code> let ys = zip str [0..] in</code>
<code> foldr (\(z,i) acc -> if (i `mod` n) /= 0 || i == 0</code>
<code> then z:acc else del ++ z:acc) [] ys</code></td></tr>
<tr><td>Constraint</td><td>Yes</td><td>No</td></tr>
<tr><td><code>Eq a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Integral a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Show a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Eq b</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Integral b</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Show b</code></td><td>◯</td><td>◯</td></tr>
</table>

<table>
<tr><td>Partial Contract</td><td colspan="2"><code>anyCloseTo :: (?) => a -> a -> [a] -> Bool</code></td></tr>
<tr><td>Function</td><td colspan="2"><code>anyCloseTo v d =</code>
<code> any ((<d).abs.((-)v))</code></td></tr>
<tr><td>Constraint</td><td>Yes</td><td>No</td></tr>
<tr><td><code>Eq a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Num a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Ord a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Fractional a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Show a</code></td><td>◯</td><td>◯</td></tr>
<tr><td><code>Read a</code></td><td>◯</td><td>◯</td></tr>
</table>

2 List Comprehension

[6 points] a) Consider the following list comprehensions featuring both generators and guards. For each of them, decide whether they are *compiling* or *not compiling*, and if they compile, write down the resulting list.

`[x*y	x <- [1,2,3], y <- "abc"]`	
not compiling ○	
compiling ○	Result:

`[(x,y)	x <- [1,2,3], x>2, y <- [2..4]]`	
not compiling ○	
compiling ○	Result:

`[tail x	x <- ["foo","bar","baz"]]`	
not compiling ○	
compiling ○	Result:

[5 points] b) Translate the following function to an equivalent one that does **not** use List Comprehension.

```
getIndicesOfUpper str = [i | (c,i) <- zip str [0..], isUpper c]
```

3 Pattern Matching

[10 points] a) Consider the following data declaration. Which of the following four expressions match the pattern given in the same table? Tick off whether a pattern matches, and if so, provide the values of the included variables. Only tick off one circle per table.

```
data RoomType = LectureHall | CIPPool deriving (Show)
data Room = Room String RoomType Int deriving (Show)
```

1)

Pattern	`(_:a)`
Expression	`"foo"`
◯ Does not match	
◯ Does match:	
`a =`	______

2)

Pattern	`[a,b]`
Expression	`"bar"`
◯ Does not match	
◯ Does match:	
`a =`	______
`b =`	______

3)

Pattern	`((_:a):b:xs)`
Expression	`[[1,2],[3,4],[5,6]]`
◯ Does not match	
◯ Does match:	
`a =`	______
`b =`	______
`xs =`	______

4)

Pattern	`(Room (_:n) _ c)`
Expression	`Room "E2.058P2" CIPPool 15`
◯ Does not match	
◯ Does match:	
`n =`	______
`c =`	______

[4 points] b) Consider the following data declaration.

```
data StudyProgram = CompEng | CompScience | TechnoMath | BioProcEng
                  | ElectricEng | EnergyEng | Logistics | MechEng
                  | Mechatronics | NavalArch | ProcEng deriving (Eq)
```

Reimplement the following `studyDetails` function using pattern matching. You are not allowed to use guards, if-else-expressions, or case-expressions.

```
studyDetails :: (StudyProgram, Int) -> String
studyDetails info
  | (snd info) == 1 = "Freshman"
  | (fst info) `elem` [CompEng, CompScience] = "Higher␣semester␣informatics"
  | otherwise = "Other␣higher␣semester"
```

4 Recursion

[6 points] a) In the following zipWith function, identify base- and recursive cases (refer to the line number), and argue under which conditions the function terminates and why (consider both finite and infinite inputs).

```
1 zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
2 zipWith f [] _ = []
3 zipWith f _ [] = []
4 zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

[4 points] b) Provide a definition for the replicate function from the Prelude using recursion, without using the original replicate function. The function should throw an exception (using error) if the replication count (first argument) is negative.

Hint: In the following, you are given the type of replicate and one example application.

```
replicate :: Int -> a -> [a]
{- Example
 - replicate 3 'a' = "aaa"
 -}
```

5 Higher-order Functions

[3 points] a) Provide a function implementation for the following type:

```
a -> b -> (a -> b -> c -> d) -> (b -> c) -> (b -> d -> e) -> e
```

[5 points] b) Provide a contract, purpose, and one example for the following function.

```
data Item = Apple | Carrot | Chocolate deriving (Eq)

advice items =
  let (hPr,uPr) = foldr (\(it,num,pr) (hAcc, uAcc) -> case it of
        Apple -> (hAcc+(num*pr), uAcc)
        Carrot -> (hAcc+(num*pr), uAcc)
        Chocolate -> (hAcc, uAcc+(num*pr))) (0,0) items in
  case hPr > uPr of
    True -> "Well␣done."
    False -> "Buy␣more␣healthy␣food!"
```

[5 points] c) The `checkData` function takes a time series (represented as a list of time-value-tuples), and both an upper and lower bound value. It returns a tuple where first value indicates if **all** values are between the bounds, and second value holds a **list of times** at which the values are outside of those bounds. Provide an implementation for `checkData` using no additional functions besides null, ||, <, >, filter, map, fst, and lambda-expressions.

6 User-defined Types

[6 points] a) Consider the following data type definitions:

```haskell
type TimeEntry = (Int, Int) -- (hour, minutes)
type DateEntry = (Int, Int, Int) -- (year, month, day)
type Year = Int
type ECTS = Int
type Professor = String
type Topic = String

data Term = WinterTerm | SummerTerm deriving Show
data SemesterData = SemesterData Year Term [Course] deriving Show

data Course = Course {
  lecturer :: Professor,
  topic :: Topic,
  ects :: ECTS,
  lectureSlots :: [LectureSlot]
} deriving Show

data LectureSlot = LectureSlot {
  date :: DateEntry,
  startTime :: TimeEntry,
  endTime :: TimeEntry,
  slotTopic :: Topic
} deriving Show
```

i) Use these data types to define the following semester data:

The semester data of the winter term of 2020 features a single 6 ECTS course "FP" by Prof. Schupp, with only one remaining lecture slot on 2020/01/29 between 9:45 and 11:15 with the topic "Revision".

ii) Decide for each of the following definitions if it is *compiling* or *not compiling*.

`Course { lecturer="Marrone", topic="ML", ects=6 }`	
compiling ◯	*not compiling* ◯

`Term WinterTerm`	
compiling ◯	*not compiling* ◯

`SemesterData 2025 SummerTerm []`	
compiling ◯	*not compiling* ◯

[6 points] b) A storage device is divided into several partitions, which have a unique mount letter and a file system. The file system features a tree structure of the data, consisting of files and directories, where the latter can contain more files and directories.

In other words, a storage device can be defined as follows:

i) MountLetter: A String type alias for the unique mount letter.

ii) Name: A String type alias for a file or directory name.

iii) Data: A String type alias for the data that every file holds.

iv) StorageDevice: A storage device consisting of multiple partitions.

v) Partition: A partition with a mount letter and a file system.

vi) FileSystem: A file system with a list of file system tree items.

vii) FileSystemTreeItem: A tree item which can either be a directory with a name and multiple sub-items, or a file with a name and data

Define the described type names and a recursive data type for the specified StorageDevice.

7 Evaluation

[6 points] a) Does the evaluation of the following Haskell expressions terminate if evaluated fully? Briefly motivate your answer (1-2 sentences).

i) `take 2 $ drop 2 $ cycle [1]`

ii) `zip [0..] [1,2,3]`

iii) `minimum [1,2..]`

[6 points] b) What is the result of a full evaluation of the following expressions?

i) `(\x -> not $ not x) True`

ii) `(\f g x -> (f x, g x)) (*2) (*3) 5`

iii) `(\f g -> \h xs -> h g xs) (+2) (+1) map [1,2,3]`

[4 points] c) Consider the three definitions of `checkPrices1`, `checkPrices2`, and `checkPrices3` below, which all take a list of purchased items and a price cap, and return whether the total price is below or equal to the cap value. The three functions differ in their concrete implementation. Considering both finite and infinite lists as possible inputs, which implementation(s) would you choose and why? Briefly (3 sentences) motivate your answer!

```
--                                item  amount  price
data ItemPurchase = ItemPurchase String Integer Float

checkPrices1, checkPrices2, checkPrices3 :: [ItemPurchase] -> Float -> Bool

checkPrices1 ps cap = priceHelper 0 ps
  where
  priceHelper total [] = total <= cap
  priceHelper total ((ItemPurchase i n p):xs) = priceHelper (total+fromInteger(n)*p) xs

checkPrices2 ps cap = (sum $ map (\\(ItemPurchase i n p) -> fromInteger(n)*p) ps) <= cap

checkPrices3 ps cap = priceHelper 0 ps
  where
  priceHelper total [] = total <= cap
  priceHelper total ((ItemPurchase i n p):xs)
    | total > cap = False
    | otherwise = priceHelper (total+fromInteger(n)*p) xs
```

8 Reasoning and Testing

[5 points] a) Consider the standard `nub` function, which takes a list `xs` of arbitrary type (derived from `Eq`) as input, and returns the sub-list `nubRet` where all duplicates elements are removed, fulfilling the following specification:

i) The number of elements in `nubRet` is smaller or equal to the number of elements in `xs`

ii) `nubRet` contains only unique elements (i.e., no duplicates)

Examples:

`nub [4,4,1,2,3,3]` ⇒ `[4,1,2,3]`

`nub "aabbbcc"` ⇒ `"abc"`

Write a generalized test for each of the two parts of the specification.

[6 points] b) In the following, you are given a definition of natural numbers, and the modulo functions `mod` and `modHelper`.

```
data Nat = Zero | Succ Nat

mod :: Nat -> Nat -> Nat
mod m n = modHelper n Zero m n

modHelper :: Nat -> Nat -> Nat -> Nat -> Nat
modHelper Zero _ a b = mod a b
modHelper _ r Zero b = r
modHelper (Succ c) r (Succ a) b = modHelper c (Succ r) a b
```

Your task is to prove for all values `y` of type `Nat` that `y` modulo one is zero. In other words, you must prove the following:

∀ y: `mod y (S Z) == Z`

Hint: Use induction on `y`! To keep the terms short, you may abbreviate `Zero` by `Z`, `Succ` by `S`, `mod` by `m`, `modHelper` by `mH`, `True` by `T`, and `False` by `F`.

Contract Selection from Prelude

```
(!!) ::  [a] -> Int -> a
($!) ::  (a -> b) -> a -> b
($) ::  (a -> b) -> a -> b
(&&) ::  Bool -> Bool -> Bool
(*) ::  Num a => a -> a -> a
(+) ::  Num a => a -> a -> a
(++) ::  [a] -> [a] -> [a]
(-) ::  Num a => a -> a -> a
(.) ::  (b -> c) -> (a -> b) -> a -> c
(/) ::  Fractional a => a -> a -> a
(/=) ::  Eq a => a -> a -> Bool
(<) ::  Ord a => a -> a -> Bool
(<=) ::  Ord a => a -> a -> Bool
(==) ::  Eq a => a -> a -> Bool
(>) ::  Ord a => a -> a -> Bool
(>=) ::  Ord a => a -> a -> Bool
(>>=) ::  Monad m => m a -> (a -> m b) -> m b
(^) ::  (Num a, Integral b) => a -> b -> a
(||) ::  Bool -> Bool -> Bool
abs ::  Num a => a -> a
all ::  (a -> Bool) -> [a] -> Bool
and ::  [Bool] -> Bool
any ::  (a -> Bool) -> [a] -> Bool
chr ::  Int -> Char
concat ::  [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
curry :: ((a, b) -> c) -> a -> b -> c
cycle ::  [a] -> [a]
div ::  Integral a => a -> a -> a
drop ::  Int -> [a] -> [a]
dropWhile ::  (a -> Bool) -> [a] -> [a]
elem ::  Eq a => a -> [a] -> Bool
even ::  Integral a => a -> Bool
filter ::  (a -> Bool) -> [a] -> [a]
foldl ::  (a -> b -> a) -> a -> [b] -> a
foldr ::  (a -> b -> b) -> b -> [a] -> b
fromInteger ::  Num a => Integer -> a
fst ::  (a, b) -> a
getChar ::  IO Char
head ::  [a] -> a
id ::  a -> a
isAsciiLower ::  Char -> Bool
isAsciiUpper ::  Char -> Bool
isLower ::  Char -> Bool
isUpper ::  Char -> Bool
length ::  [a] -> Int
lines ::  String -> [String]
map ::  (a -> b) -> [a] -> [b]
max ::  Ord a => a -> a -> a
maximum ::  Ord a => [a] -> a
minimum ::  Ord a => [a] -> a
mod ::  Integral a => a -> a -> a
negate ::  Num a => a -> a
not ::  Bool -> Bool
nub ::  Eq a => [a] -> [a]
null ::  [a] -> Bool
odd ::  Integral a => a -> Bool
or ::  [Bool] -> Bool
ord ::  Char -> Int
otherwise ::  Bool
product ::  Num a => [a] -> a
```

```
putChar :: Char -> IO ()
putStr :: String -> IO ()
putStrLn :: String -> IO ()
read :: Read a => String -> a
repeat :: a -> [a]
replicate :: Int -> a -> [a]
return :: Monad m => a -> m a
reverse :: [a] -> [a]
seq :: a -> b -> b
show :: Show a => a -> String
signum :: Num a => a -> a
snd :: (a, b) -> b
sqrt :: Floating a => a -> a
sum :: Num a => [a] -> a
tail :: [a] -> [a]
take :: Int -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
toLower :: Char -> Char
toUpper :: Char -> Char
transpose :: [[a]] -> [[a]]
uncurry :: (a -> b -> c) -> (a, b) -> c
union :: Eq a => [a] -> [a] -> [a]
unlines :: [String] -> String
unzip :: [(a, b)] -> ([a], [b])
zip :: [a] -> [b] -> [(a, b)]
```