Eexam
Place student sticker here

Note:
- During the attendance check a sticker containing a unique code will be put on this exam.
- This code contains a unique number that associates this exam with your registration number.
- This number is printed both next to the code and to the signature field in the attendance check list.

Funktionale Programmierung und Verifikation

Exam:	IN0003 / Endterm	**Date:**	Wednesday 8th January, 2020
Examiner:	Prof. Tobias Nipkow, Ph.D.	**Time:**	13:00 – 15:00

	P 1	P 2	P 3	P 4	P 5	P 6	P 7	P 8
I								

Working instructions

- This exam consists of **16 pages** with a total of **8 problems**.
  Please make sure now that you received a complete copy of the exam.
- The total amount of achievable credits in this exam is 40 credits.
- Detaching pages from the exam is prohibited.
- Allowed resources:
  - one **handwritten** sheet of A4 paper
  - one **analog dictionary** English ↔ native language **without annotations**
- Subproblems marked by * can be solved without results of previous subproblems.
- Do not write with red or green colors nor use pencils.
- Physically turn off all electronic devices, put them into your bag and close the bag.

Left room from ________ to ________ / Early submission at ________

Problem 1 **Type Inference (5 credits)**

a)* Determine the most general type of these expressions.

1. `(:[1,2])`
2. `foldr (\x y -> y ++ x) []` (where `foldr :: (a -> b -> b) -> b -> [a] -> b`)
3. `(\f g x -> g $ f $ x)`
4. `map head . map (\f -> f "hello")`

b)* Give a brief justification why these expressions do not type check.

1. `(\f x -> if True then f x else f)`
2. `(\x y -> x : y : x y)`

Problem 2 List Comprehension, Recursion, Higher Order Functions (6 credits)

Write a function `halfEven :: [Int] -> [Int] -> [Int]` that takes two lists xs and ys as input. The function should compute the pairwise sums of the elements of xs and ys, i.e. for $xs = [x_0, x_1, \ldots]$ and $ys = [y_0, y_1, \ldots]$ it computes $[x_0 + y_0, x_1 + y_1, \ldots]$. Then, if the sum is even, the sum is halved and added to the resulting list. An invocation of `halfEven` could look like follows:

```
halfEven [1, 2, 3, 4] [5, 3, 1] = [3, 2]
```

Implement the function in three different ways:

1. As a list comprehension without using any higher-order functions or recursion.
2. As a recursive function with the help of pattern matching. You are not allowed to use list comprehensions or functions from the Haskell library.
3. Use higher order functions (e.g. map, filter, etc.) but no recursion or list comprehensions.

0 1 2 3 4 5 6

Problem 3 Obligatory Logic Exercise (6 credits)

We define the following types:

- An *atom* is either `F` (falsity), `T` (truth), or a variable:

```
type Name = String
data Atom = F | T | V Name deriving (Eq, Show)
```

- A *conjunction* is an atom or the conjunction of two conjunctions:

```
data Conj = A Atom | Conj :&: Conj deriving (Eq, Show)
```

- A *database* is a finite set of atoms that we simply model as a conjunction:

```
type Db = Conj
```

  For example, the database $\{"v", "w"\}$ can be modelled as `A (V "v") :&: A (V "w")`.

0 1 2

a)* Write a function `contains :: Db -> Atom -> Bool` such that `contains db a` returns `True` if and only if `a` is contained in `db`.

0 1 2 3 4

b)* Write a function `implConj :: Conj -> Conj -> Bool` such that `implConj c1 c2` returns `True` if and only if conjunction `c1` logically implies conjunction `c2`. For example:

```
A F `implConj` c = True -- for any conjunction c
c `implConj` A T = True -- for any conjunction c
A (V "v") `implConj` A (V "v") = True
A (V "v") `implConj` A (V "v") :&: A (V "w") = False
A (V "w") :&: A (V "v") `implConj` A (V "v") :&: A (V "w") = True
```

Problem 4 Haskell has Class (5 credits)

We define a typeclass of integer containers as follows:

```
class IntContainer c where
  -- the empty container
  empty :: c
  -- insert an integer into a container
  insert :: Integer -> c -> c
```

Moreover, we define an extension of integer containers called `IntCollection` as follows:

```
class IntContainer c => IntCollection c where
  -- the number of integers in the collection
  size :: c -> Integer
  -- True if and only if the integer is contained in the collection
  contains :: Integer -> c -> Bool
  -- c1 `subcolleq` c2 holds if and only if
  -- every integer in c1 also occurs in c2
  subcolleq :: c -> c -> Bool
  -- extracts the smallest number in the collection
  -- if such a number exists.
  extractMin :: c -> Maybe Integer
  -- "update f c" applies f to every element e of c.
  -- If "f c" returns Nothing, the element is deleted;
  -- otherwise, the new value is stored in place of e.
  update :: (Integer -> Maybe Integer) -> c -> c
```

Assume there is a type `data C` with a corresponding `IntContainer` instance. Moreover, assume you are given the following function:

```
-- "fold f acc c" folds the function f along c (in no particular order)
-- using the start accumulator acc.
fold :: (Integer -> b -> b) -> b -> c -> b
```

Define an instance `IntCollection C`.

0
1
2
3
4
5

Problem 5 Proof 1 (4 credits)

0
1
2
3
4

Given the type of natural numbers

```
data Nat = Z | Suc Nat
```

and the following definition of addition on these numbers

```
add Z m = m
add (Suc n) m = Suc (add n m)
```

show that addition is associative by proving this equation

```
add (add x y) z  = add x (add y z)
```

Problem 6 Proof 2 (5 credits)

You are given the following definitions:

```
data Tree a = L | N (Tree a) a (Tree a)

flat :: Tree a -> [a]
flat L = []
flat (N l x r) = flat l ++ (x : flat r)

app :: Tree a -> [a] -> [a]
app L xs = xs
app (N l x r) xs = app l (x : app r xs)

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

Prove the following lemma:

```
app t [] = flat t
```

You may use associativity of `++` in the proof.
Hint you may have to generalize the lemma first.

Problem 7 IO (6 credits)

0
1
2
3
4
5
6

Define an IO action `main :: IO ()` that waits for user input in form of a binary number. The binary number is given as a string 0*bx* where x is (potentially empty) string consisting of 0s and 1s. The string 0*b* represents 0. The program should output `Invalid input` if the given number does not adhere to this format. Otherwise, the program should print the number to the standard output after converting it to decimal. For example, the program should output 5 for the input 0*b*0101. The program should continue to listen for the next input in either of the above cases.

You can read from standard input with the function `getLine :: IO String` and print a string to the standard output with `putStrLn :: String -> IO ()`.

Problem 8 **Evaluation (3 credits)**

Evaluate the following expressions as far as possible using the evaluation strategy described in the lecture. Indicate infinite reductions by "...", as soon as nontermination becomes apparent.

0
1
2
3

1. `take 2 odds`

2. `False || inf == inf`

3. `(\f -> \g -> g . map f) (+1) head odds`

```
take :: Integer -> [a] -> [a]
take 0 _ = []
take n (x:xs) = x : take (n-1) xs

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

odds :: [Integer]
odds = 1 : map (+2) odds

head :: [a] -> a
head (x:xs) = x
head _ = error("empty list")

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

(||) :: Bool -> Bool -> Bool
True || b = True
False || b = b

inf :: a
inf = inf
```

Additional space for solutions–clearly mark the (sub)problem your answers are related to and strike out invalid solutions.