Chair of Logic and Verification
Department of Informatics
Technical University of Munich

Esolution
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:**	Saturday 8th February, 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**
- You may answer in **German** or **English**.
- 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)

0 a) Determine the most general type of these expressions:

1 1. `foldr (\x y -> y ++ x) []` (where `foldr :: (a -> b -> b) -> b -> [a] -> b`)

2 2. `(\f g x -> g $ f $ x)`

3 3. `(:[1,2])`

4 4. `map head . map (\f -> f "hello")`

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

1 1. `if f x then x else "error"` (where `f :: (a -> Bool)` and `x :: Int`)

2. `1 : 2 : f x` (where `f :: (a -> String)` and `x :: Int`)

a)

1. `[[a]] -> [a]`
2. `(a -> b) -> (b -> c) -> a -> c`
3. `Num a => a -> [a]`
4. `[String -> [a]] -> [a]`

b)

1. Then branch returns `Int`, else branch `String`
2. `f x :: String`, there is no instance of `Num` for `Char`

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 $x_i + y_i$ is even, the sum is halved. Otherwise, the sum is removed from the list. An invocation of `halfEven` could look as follows:

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

Implement the function in three different ways:

a) As a list comprehension without using any higher-order functions or recursion.

```
halfEven xs ys = [(x + y) `div` 2 | (x, y) <- zip xs ys, even (x + y)]
```

b) As a recursive function with the help of pattern matching. You are not allowed to use list comprehensions or higher-order functions.

```
halfEven [] _ = []
halfEven _ [] = []
halfEven (x:xs) (y:ys)
  | even (x + y) = ((x + y) `div` 2) : halfEven xs ys
  | otherwise = halfEven xs ys
```

c) Use higher-order functions (e.g. map, filter, etc.) but no recursion or list comprehensions.

```
halfEven xs ys = map (flip div 2) . filter even . map (uncurry (+)) $ zip xs ys
```

Problem 3 **Obligatory Logic Exercise (5 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)
```

0 1

a) Write a function `contains :: Conj -> Atom -> Bool` such that `contains c a` returns `True` if and only if `a` occurs in `c`.

```
contains :: Conj -> Atom -> Bool
contains (A a) a' = a == a'
contains (c1 :&: c2) a = contains c1 a || contains c2 a
```

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
```

```
implConj :: Conj -> Conj -> Bool
implConj c (A a) = a == T || contains c F || contains c a
implConj c (c1 :&: c2) = implConj c c1 && implConj c c2

-- Alternative solution
-- implAtom :: Conj -> Atom -> Bool
-- implAtom _ T = True
-- implAtom (A F) _ = True
-- implAtom (A a) a' = a == a'
-- implAtom (c1 :&: c2) a = implAtom c1 a || implAtom c2 a

-- implConj :: Conj -> Conj -> Bool
-- implConj c (A a) = implAtom c a
-- implConj c (c1 :&: c2) = implConj c c1 && implConj c c2
```

Problem 4 Haskell Has Class (5.5 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 a member of the collection
  member :: Integer -> 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
  -- "partition p c" creates two collections (c1,c2) such that
  -- c1 contains exactly those elements of c satisfying p and
  -- c2 contains exactly those elements of c not satisfying p.
  partition :: (Integer -> Bool) -> c -> (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

```
instance IntCollection C where
  size = fold (const (+1)) 0
  member x = fold ((||) . (==x)) False
  extractMin = fold aux Nothing
    where aux x Nothing = Just x
          aux x (Just y) = Just (min x y)
  update f = fold (aux . f) empty
    where aux Nothing  = id
          aux (Just x) = insert x
  partition p = fold aux (empty, empty)
    where aux x (c1,c2)
            | p x = (insert x c1, c2)
            | otherwise = (c1, insert x c2)
```

Problem 5 Wishes From Peano (4 credits)

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 the following equation using structural induction:

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

0 1 2 3 4

```
Lemma: add (add x y) z .=. add x (add y z)
Proof by induction on x
Case Z
  To show: add (add Z y) z .=. add Z (add y z)

                     add (add Z y) z
    (by def add) .=. add y z
    (by def add) .=. add Z (add y z)

Case (Suc x)
  To show: add (add (Suc x) y) z .=. add (Suc x) (add y z)
  IH: add (add x y) z .=. add x (add y z)
                     add (add (Suc x) y) z
    (by def add) .=. add (Suc (add x y)) z
    (by def add) .=. Suc (add (add x y) z)
    (by IH)      .=. Suc (add x (add y z))
    (by def add) .=. add (Suc x) (add y z)

QED
```

Problem 6 Proof 2 (5 credits)

0 1 2 3 4 5

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 statement using structural induction:

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

You may use the following lemmas about `++` in the proof:

```
Lemma ++_assoc: (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
Lemma ++_nil: xs ++ [] = xs
Lemma nil_++: [] ++ xs = xs
```

Hint: you should generalize the statement first.

We generalize the property `app t [] = flat t` to the following statement:

```
Lemma gen: app t xs = flat t ++ xs
Proof by induction on Tree t
Case L:
  To show: app L xs = flat L ++ xs

                    app L xs
    (def app)    = xs
    (def ++)     = [] ++ xs
    (def flat)   = flat L ++ xs

Case (N l x r):
  To show: app (N l x r) xs = flat (N l x r) ++ xs
  IH1: app l xs = flat l ++ xs
  IH2: app r xs = flat r ++ xs

                    app (N l x r) xs
    (def app)    = app l (x : app r xs)
    (by IH1)     = flat l ++ (x : app r xs)
    (by IH2)     = flat l ++ (x : (flat r ++ xs))

                       flat (N l x r) ++ xs
    (def flat)       = (flat l ++ (x : flat r)) ++ xs
    (by ++_assoc)    = flat l ++ ((x : flat r) ++ xs)
    (def ++)         = flat l ++ (x : (flat r ++ xs))

QED
```

```
Our goal then follows:

Lemma: app t [] = flat t
Proof
                      app t []
    (by gen)       = flat t ++ []
    (by ++_nil)  = flat t
QED
```

Problem 7 IO (6.5 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 $0bx$ where x is a (potentially empty) string consisting of 0s and 1s. The string $0b$ 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 $0b0101$. The program should continue to listen for the next input in either of the above cases. As an example, consider the following excerpt of the execution of the program.

```
>>> 0b12
Invalid input
>>> 0b010
2
>>> 0b111
7
...
```

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

```
toDecimal :: String -> Integer -> Integer
toDecimal [] _ = 0
toDecimal ('0':bs) r = toDecimal bs (r * 2)
toDecimal ('1':bs) r = r + toDecimal bs (r * 2)


main :: IO ()
main = do
  bin <- getLine
  let (pref, num) = splitAt 2 bin
  if pref /= "0b" || any (\b -> b `notElem` ['0','1']) num
    then putStrLn "Invalid input"
    else print $ toDecimal (reverse num) 1
  main
```

Problem 8 Evaluation (3 credits)

Given the following definitions:

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

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

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

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

inf :: [a]
inf = inf

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

Using Haskell's evaluation strategy as introduced in the lecture, evaluate the following expressions step-by-step as far as possible . Indicate infinite reductions by "..." as soon as nontermination becomes apparent.

0 1 2 3

1. `(\f g -> g . map f) (+1) head odds`
2. `False || inf == inf`

1.

```
(\f -> \g -> g . map f) (+1) head odds
(\g -> g . map (+1)) head odds
(head . map (+1)) odds
(\x -> head (map (+1) x)) odds
head (map (+1) odds)
head (map (+1) (1 : map (+2) odds))
head (((+1) 1) : map (+1) (map (+2) odds))
(+1) 1
2
```

2.

```
False || inf == inf
inf == inf
inf == inf
...
```

Sample Solution

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