Dauer	1h 50m
Maximale Punktezahl	90
Benötigte Punktzahl	>= 70%: sehr gut; >= 60% gut; >= 50% befriedigend; >= 40% genügend; < 40%: nicht genügend

Analysis of Programs (5 points, single choice)

Question

Lösung

Consider the following three Haskell definitions of function `f`.

```
-- definition 1
f xs =
    (xs == [])

-- definition 2
f (_ : _) = False
f [] = True

-- definition 3
f [] = True
f _  = False
```

We say that two function definitions are equivalent, whenever in any program one can replace the one definition by the other without changing the behavior of the program, i.e., the success of program compilation is independent of the choice, as well as the input-output-behavior.

- [✓ a. Definition 3 is equivalent to 2, but 1 is different.]
- b. Definition 2 is equivalent to 1, but 3 is different.
- c. Definition 3 is equivalent to 1, but 2 is different.
- d. None of the definitions are equivalent.
- e. All three definitions are equivalent.

Zurück zur Übersicht

Higher-Order Functions (5 points, single choice)

Question

Lösung

Consider the following function:

```
bar :: (a -> a) -> [a -> a] -> (a -> a)
bar = foldr (\ z r -> r)
```

The expression `bar f0 [f1, ..., f9] x` has the same value as:

- a. `f9 (f8 (f7 ... (f0 x) ... ))`
- b. None of the other answers is correct.
- c. `f0 (f1 (f2 ... (f9 x) ... ))`
- [✓ d. `f0 x`]
- e. `f9 x`

Zurück zur Übersicht

Polymorphism (5 points, single choice)

Question

Lösung

In Haskell, function definitions can be polymorphic, e.g., the identity function `id` has a polynmorphic type where the input and the return type are identical, namely of type `a`.

```
id :: a -> a
id x = x
```

Is it possible to define a Haskell function `f :: a -> b` where the input and return type are different type variables?

- ○ a. Yes, this is possible, but only with the help of the predefined functions `undefined` and `error`.
- [✓ b. Yes, this is possible, even without using the predefined functions `undefined` and `error` and without adding a type-constraint on `b`.]
- ○ c. No, this is not possible.
- ○ d. Yes, this is possible, even without using the predefined functions `undefined` and `error`, but one has to add a type-constraint on `b`.

Zurück zur Übersicht

Programming with Numbers and Lists (40 points)

Question

Lösung

For the whole exercise, it is not allowed to use any imports.

Write your solutions for all programming tasks below into a single Haskell-file and upload it. The file must be compilable. Use comments or `undefined` to quickly turn a non-compilable Haskell file into a compilable one, e.g. via `someFunction = undefined`.

In this exercise, you have to implement algorithms where natural numbers are represented in binary, i.e., as list of bits (0 and 1). For instance,

- 0 is represented as the list [0],
- 6 is represented as the list [1,1,0], since $6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$.
- 11 is represented as the list [1,0,1,1], since $11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$.

We always assume that non-zero numbers are represented without leading zeros, i.e., [0,0,1,1,0] is not a valid representation of 6, since the two leading zeros are not permitted. Hence, you can always assume that input binary numbers don't have leading zeros, but it is your task to make sure that your generated binary numbers do not produce leading zeros.

To represent binary natural numbers in Haskell, the following datatype has been created, and the numbers 2 and 11 are encoded.

```haskell
data Bit = One | Zero deriving (Show, Eq)
type Nat = [Bit]

two :: Nat
two = [One,Zero]

eleven :: Nat
eleven = [One,Zero,One,One]
```

Most of the following tasks can be solved independently.

Hint: For some of the tasks it might be beneficial to reverse some lists of bits.

Task 1 (10 points)

Define a function

```haskell
natToInt :: Nat -> Integer
```

such that `natToInt` converts a number in binary representation into the same number of type integer.

Example: `natToInt eleven` evaluates to 11.

Task 2 (10 points)

Define a function `intToNat :: Integer -> Nat` which converts a non-negative integer into binary representation. Note that leading zeros are not permitted.

Example: `intToNat 6 = [One,One,Zero]`.

Hint: `div x 2` and `mod x 2` are your friend.

Task 3 (8 points)

Implement a function `fullAdder :: Bit -> Bit -> Bit -> (Bit, Bit)` which adds up three bits, i.e., whenever

```haskell
fullAdder a b c = (sum, carry)
```

then

- the `carry` is 1 if and only if at least two bits of `a,b,c` are 1, and
- `sum = (a + b + c) mod 2`

For example, `fullAdder One Zero One = (Zero, One)`.

Task 4 (12 points)

Define a function `addNat :: Nat -> Nat -> Nat` which implements addition of two binary numbers. Your implementation should use `fullAdder` of Task 3, but it is not allowed to use Haskell's built-in addition function `(+) :: Integer -> Integer -> Integer`.

Example: `addNat eleven two` should result in `[One,One,Zero,One]`, and here are the intermediate values that you get via the standard addition algorithm from school.

```
x = 11:      1 0 1 1
y =  2:          1 0
carry:     0 0 1 0
--------------------
sum = 13:    1 1 0 1
```

Zurück zur Übersicht

Programming with List Comprehensions (15 points)

Question

▾ Lösung

For the whole exercise, it is not allowed to use any imports.

Write your solutions for all programming tasks below into a single Haskell-file and upload it. The file must be compilable. Use comments or `undefined` to quickly turn a non-compilable Haskell file into a compilable one, e.g. via `someFunction = undefined`.

As a common part we consider an inventory list. This list contains triples, where each triple consists of a name of a product, the price of a single product, and the stored quantity of that product.

In Haskell this is modeled as follows, and an example inventory list of a tiny book store is defined.

```
type InventoryList = [(String,Double,Integer)]

bookStore :: InventoryList
bookStore = [
     ("Pilates for beginners", 19.90, 20),
     ("Harry Potter 2021", 29.90, 0),
     ("Sharks in the Inn", 8.90, 15),
     ("Dolphins and wales", 15.99, 50)
  ]
```

So, in the example book store, three books are available, and there has been an announcement of a future book that is not yet available, i.e., the quantity may be 0, but it is never negative.

Hint: All of the following tasks can be (but don't have to be) solved with a one-line program by using list-comprehensions or functions on lists that have been presented in the lecture.

Task 1 (5 points)

Define a function `boundProducts :: InventoryList -> Double -> Double -> [String]`. It should take a lower price limit and an upper price limit and return the names of all products that fall within the price range and are available.

Example: `boundProducts bookStore 10 30 = ["Pilates for beginners","Dolphins and wales"]`

Task 2 (5 points)

Write a function `totalValue :: InventoryList -> Double` that computes the total value of all products in the store.

Example: `totalValue bookStore = 19.90 * 20 + ... + 15.99 * 50 = 1331.0`.

Task 3 (5 points)

Define a function `expensiveProducts :: InventoryList -> [String]` that identifies the names of the most expensive products (ignoring availability).

Example: In the book store, the most expensive product is "Harry Potter 2021".

Note that in general there can be several products that have the maximal price.

‹ Zurück zur Übersicht

Programming with Input and Output (20 points)

Question

▾ Lösung

For the whole exercise, it is not allowed to use any imports.

Write your solutions for all programming tasks below into a single Haskell-file and upload it. The file must be compilable. Use comments or `undefined` to quickly turn a non-compilable Haskell file into a compilable one, e.g. via `someFunction = undefined`.

Task 1 (7 points)

Define a function `isNumber :: String -> Bool` that tests whether a given string encodes an integer number, i.e., any string that purely consists of digits 0,...,9, at least one of them, and may optionally have a negation sign '-' in front.

Example: `map isNumber ["-0049", "", "0x73", "876"] = [True, False, False, True]`

Task 2 (13 points)

Define a Haskell program that contains a function `main :: IO ()`. When executed the program should print an initial text and then read one line after the other. It should stop its execution when reading a line that does not contain an integer number, and then prints the sum of all numbers that have been read before.

Here is an example dialog. The first and last line have been printed by the program, the remaining lines have been entered by the user.

```
Enter some numbers.
-3
-1
-2
give me the result
The sum of the numbers is -6.
```

Make sure that in your program, the output is precisely formatted as indicated.