Programmier-Paradigmen

Tutorium – Gruppe 4 & 8

Henning Dieterichs

Blatt 1 – pow1, pow2, pow3

```haskell
module Arithmetik where
    
    pow1 :: Int → Int → Int
    pow1 b 0 = 1
    pow1 b e = b * pow1 b (e - 1)
    
    pow2 :: Int → Int → Int
    pow2 b 0 = 1
    pow2 b e
        | even e = pow2 (b*b) (e `div` 2)
        | otherwise = b * pow2 (b*b) (e `div` 2)

    pow3 :: Int → Int → Int
    pow3 = pow3Acc 1
        where
            pow3Acc acc b 0 = acc
            pow3Acc acc b e
                | e < 0 = error "Negative exponent"
                | even e = pow3Acc acc (b*b) (e `div` 2)
                | otherwise = pow3Acc (acc * b) (b*b) (e `div` 2)
```

Blatt 1 – isPrime

```haskell
module Arithmetik where
    
    isPrime :: Int → Bool
    isPrime num
        | num < 0 = error "Not a natural number"
        | num == 1 = False
        | otherwise = null [ x | x ← [2..root 2 num], num `mod` x == 0 ]

    isPrime' :: Int → Bool
    isPrime' num
        | num < 0 = error "Not a natural number"
        | num == 1 = True
        | otherwise = not $ isNumDivisibleByAnyNumberIn 2 $ root 2 num
        where
            -- checks whether num is divisible by any nat. number in [a, b]
            isNumDivisibleByAnyNumberIn a b
                | a > b = False
                | otherwise = num `mod` a == 0 || isNumDivisibleByAnyNumberIn (a+1) b

```

Blatt 1 – root

- Aufgabe: Schreibe Funktion *invert*, die eine ganzzahlige monotone Funktion auf einem Intervall invertiert

Blatt 1 – insertSort

- Aufgabe: Schreibe insert und insertSort mithilfe von insert

Blatt 1 – insertSort

```haskell
insert :: [Int] → Int → [Int]
insert (x:xs) e
    | e < x = e:x:xs
    | otherwise = x:insert e xs

insertSort :: [Int] → [Int]
insertSort = foldl insert []
```

```
insertSort [5,3,7]
⇒ foldl insert [] [5,3,7]
⇒ foldl insert (insert [] 5) [3,7]
⇒ foldl insert [5] [3,7]
⇒ foldl insert (insert [5] 3) [7]
⇒ foldl insert [3, 5] [7]
⇒ foldl insert (insert [3,5] 7) []
⇒ foldl insert [3,5,7] []
⇒ [3,5,7]
```

Blatt 1 – mergeSort

```haskell
module Sort where
    ····
    ····merge :: Ord a ⇒ [a] → [a] → [a]
    ····merge [] [] = []
    ····merge [] list = list
    ····merge list [] = list
    ····merge (head1:tail1) (head2:tail2)
    ········| head1 ⩽ head2 = head1 : merge tail1 (head2:tail2)
    ········| otherwise = head2 : merge (head1:tail1) tail2
    ····
    ····mergeSort :: Ord a ⇒ [a] → [a]
    ····mergeSort [] = []
    ····mergeSort [x] = [x]
    ····mergeSort list = merge
    ············(mergeSort $ take half list)
    ············(mergeSort $ drop half list)
    ········where half = length list `div` 2

```

Vorgriff:
Algebraische Typen

Aufgabe zum Nachdenken

- Mini Computer Algebra System
  - `Schreibe algebraischen Typ Term.`
    - `Es gibt Mult, Add, Minus, Const und Var.`
  - `Schreibe showTerm :: Int -> Term -> String`
    - `Java ToString`
  - `Schreibe differentiate :: String {-for-} -> Term -> Term`
  - `Schreibe simplify :: Term -> Term (schwer!)`