1. Übung

Börge Mehlhorn

7. April 2020

Aufgabe 1

```
uebung1.hs
-----------
#!/usr/bin/ghci
-- Aufgabe 1
sum3 :: Int -> Int -> Int -> Int
sum3 x y z = x + y + z
```

`:type`:
zeigt den Datentyp eines Audrucks. Zum Beispiel:

```
*Main> :type sum3
sum3 :: Int -> Int -> Int -> Int
*Main> :type sum3 1 2 3
sum3 1 2 3 :: Int
```

`:info`:
zeigt Informationen über einen Namen. Das sind, abhängig vom Namen, der Datentyp, der Ort der Definition usw.

```
*Main> :info sum3
sum3 :: Int -> Int -> Int -> Int    -- Defined at uebung1.hs:3:1
*Main> :info Int
data Int = GHC.Types.I# GHC.Prim.Int#   -- Defined in ‘GHC.Types’
instance Eq Int -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
[...]
*Main> :info +
class Num a where
  (+) :: a -> a -> a
  ...
    -- Defined in ‘GHC.Num’
infixl 6 +
```

`:browse`
zeigt die Namen, die durch ein Modul definiert werden und deren Typen.

```
ghci> :browse Main
sum3 :: Int -> Int -> Int -> Int
ghci> :browse GHC.Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
[...]
```

`:?`
ist ein Alias für `:help`.

Aufgabe 2

```
uebung1.hs
-----------
[...]
-- Aufgabe 2 (a)
fac :: Int -> Int
fac n = n * (if n == 1 then 1 else (fac (n - 1)))

-- Aufgabe 2 (b)
sumFacs :: Int -> Int -> Int
sumFacs n m = (fac m) + (if n == m then 0 else sumFacs n (m - 1))
```

Anmerkung:
Die Funktion `fac` liefert nur in einem Definitionsbereich von $[1, 20] \subset \mathbb{N}$ das erwartete Ergebnis, da der datentyp `Int` beschränkt ist. Entsprechend ist auch der Definitionsbereich von `sumFacs` beschränkt.

Will man für gr ößere Definitionsbereiche rechnen, kann statt `Int` der unbeschränkte Datentyp `Integer` genutzt werden.

```
ghci> maxBound :: Int
9223372036854775807
```

Aufgabe 3

```
uebung1.hs
-----------
[...]
-- Aufgabe 3
--   Implementation nach Definition
```

```haskell
fib' :: Int -> Int
fib' i = (if i > 1 then (fib' (i - 1)) + (fib' (i - 2)) else 1)
```

Anmerkung
Für größere Zahlen wird die Berechnung ineffizient. Mit der Formel von Moivre-Binet ist eine schnellere, aber auch ungenauere, Berechnung möglich.

```haskell
uebung1.hs
-----------
[...]
--  Implementation nach Moivre-Binet
sqrt5 :: Float
sqrt5 = 5 ** 0.5
phi = ((1 + sqrt5 ) / 2)
psi = 1 - phi
fib :: Float -> Float
fib i  = (((phi ** (i + 1)) - (psi ** (i + 1))) / sqrt5)
```

Die Ergebnisse:

```
*Main> fib' 0
1
*Main> fib' 1
1
*Main> fib' 3
3
*Main> fib' 4
5
*Main> fib' 8
34
*Main> fib' 30
1346269
```

```
*Main> fib 0
1.0
*Main> fib 1
1.0
*Main> fib 3
3.0
*Main> fib 4
5.0000005
*Main> fib 8
34.0
*Main> fib 30
1346269.4
```

Zusatzaufgabe 1

Nach der Definition der Aufgabe können keine Binärbäume mit gerader Knotenzahl existieren.

Wenn z die Knotenzahl ist, beschreibt die Catalan-Zahl C_n mit n gegeben durch

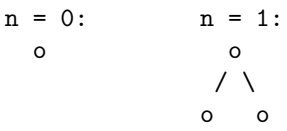
$$z = 2n + 1$$

die Zahl der möglichen Bäume.

Die Catalan-Zahl kann zum Beispiel durch folgende rekursive Vorschrift berechnet werden:

$$C_{n+1} = \frac{4n+2}{n+2} C_n$$

Für $z = 1$ und $z = 3$ ($n = 0$ und $n = 1$) ist die Zahl der möglichen Bäume trivial eins.



Die Startwerte für die Rekursion sind also

$$C_0 = C_1 = 1.$$

Somit ergibt sich folgende Haskell-Implementierung

uebung1.hs

```
----------
[...]
-- Zusatzaufgabe 1
--   Catalan-Zahl rekursiv
cat :: Rational -> Rational
cat n = (if n > 1 then (((4 * n) - 2) / (n + 1)) * cat' (n - 1)
↪  else 1)
--   z ... Anzahl der Knoten
--   z = 2n + 1
numTrees :: Int -> Integer
numTrees z = round (if z `mod` 2 == 0 then 0 else  cat
↪  ((fromIntegral z - 1) / 2))
```