HASKELL 2: Kurze Programmieraufgaben in Haskell

panitz

Zusammenfassung

Dieser Kurs besteht aus 10 kleinen Programmieraufgaben in der Programmiersprache Haskell. Es geht um weniger um Rekursion, sondern einfache Datentypen und Instanze von Typklassen.

Frage: Minuten auf Zeit hinzurechnen

Implementieren Sie für den Datentyp `Zeit` Haskell

```haskell
      module Zeit where
data Zeit
  = Zeit {stunden::Int, minuten::Int, sekunden::Int}
    deriving (Show,Eq)

minutenSpaeter z _ = z
```

Haskell

Lösung

Haskell

```haskell
module Zeit where

data Zeit
  = Zeit {stunden::Int, minuten::Int, sekunden::Int}
    deriving (Show,Eq)

minutenSpaeter (Zeit s m1 sec) m2
    = Zeit stunden (mins `mod` 60) sec
  where
    mins = m1 + m2
    stunden = (s+mins `div` 60) `mod` 24
```

Haskell

Erläuterung

Mit Division und Modulorechnung sind die Überläufe nach 60 Minuten und 24 Stunden zu ermitteln. Lokale Variablen lassen sich in einer `where`-Klausel definieren.

Frage: Test auf Schaltjahr

Implementieren Sie für den Datentyp `Datum` eine Funktion, die testet ob es sich um ein Schaltjahr handet. Haskell

```haskell
      module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Show,Eq)

schaltjahr:: Datum -> Bool
schaltjahr _ = False
```

Haskell

Lösung

Haskell

```haskell
        module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Show,Eq)

schaltjahr:: Datum -> Bool
schaltjahr (Datum _ _ j)
  |j `mod` 400 == 0 = True
  |j `mod` 4 ==0 =  j `mod `100 /= 0
  |otherwise = False
```

Haskell

Erläuterung

Per *Guards* kann man die zu verodernden Bedingungen definieren. Alle 400 Jahre ist ein Schaltjahr, ansonsten zu allen durch vier teilbaren Jahren, die nicht durch hindert teilbar sind.

Frage: Show auf Datumsobjekten

Implementieren Sie für den Datentyp `Datum` die Typklasse `Show`, so dass ein Datum im Format »17.4.2018« dargestellt wird. Haskell

```haskell
      module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Eq)

instance Show Datum where
  show  _  = ""
```

Haskell

Lösung

Haskell

```haskell
        module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Eq)

instance Show Datum where
  show (Datum t m j)
    = show t++"."++show m++"."++show j
```

Haskell

Erläuterung

Es muss die funktion `show` definiert werden. Die Zahjlen müssen ihrerseits mit der Funktion `show` zu String gemacht werden. Strings sind wiederum nur Listen, die mit ++ aneinander gehängt werden können.

Frage: Ordnung auf Datumsobjekten

Implementieren Sie für den Datentyp `Datum` die Typklasse `Ord`. Haskell

```haskell
      module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Show,Eq)

instance Ord Datum where
  _ <= _ = False
```

Haskell

Lösung

Haskell

```haskell
        module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Show,Eq)

instance Ord Datum where
  (Datum t1 m1 j1) <=  (Datum t2 m2 j2)
    |j1 < j2 = True
    |j1 == j2 && m1 < m2 = True
    |otherwise = j1==j2 && m1==m2 && t1<=t2
```

Haskell

Erläuterung

Es reicht die kleiner-gleich Funktion zu definieren. Über *pattern matching* und mit *Guards* lassen sich die entsprechenden Fälle deklarieren.

Frage: Wochentagsberechnung

Sei ein Datum *t.m.j* gegeben:
Es sei $y = j - 1$ für $m < 3$ und $y = j$ sonst.
Dann ist der Wochentag zu berechnen nach:

$$(t + (26 * ((m + 9)\ mod\ 12 + 1) - 2)/10 + y\ mod\ 100 + (y\ mod\ 100/4) + (y/400) - 2 * (y/100))\ mod\ 7$$

Die Wochentage sind dabei beginnend mit 0 für den Sonntag durchnummeriert.
Implementieren Sie diese Wochentagsberechnung. Haskell

```haskell
      module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Show,Eq)

data Tag = So|Mo|Di|Mi|Do|Fr|Sa
  deriving (Eq,Ord,Enum,Show)

wochentag :: Datum -> Tag
wochentag _ = So
```

Haskell

Lösung

Haskell

```haskell
        module Datum where
data Datum =
  Datum {tag::Int, monat::Int, jahr::Int}
    deriving (Show,Eq)

data Tag = So|Mo|Di|Mi|Do|Fr|Sa
  deriving (Eq,Ord,Enum,Show)

wochentag :: Datum -> Tag
wochentag (Datum d m j)
  = toEnum (
    (    d
     + (26*((m + 9)`mod`12 + 1) -2) `div` 10
     + y `mod` 100
     + (y `mod` 100 `div` 4)
     + (y `div` 400)
     - 2 * (y `div` 100) ) `mod` 7)
  where
    y = if (m<3) then j-1 else j
```

Haskell

Erläuterung

Zum Glück lässt sich die Formel mehr oder weniger direkt abschreiben.
Die so berechnete Zahl kann mit `toEnum` zu einen Tag umgerechnet werden.

Frage: Wochentagsberechnung

Die n-te Fibonaccizahl lässt sich auch durch folgende Formel berechnen.

$$\frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}$$

Setzen Sie die Funktion mit dieser Formel um:
Haskell

```haskell
      module Fib where
fib :: (Integral a, Integral b) => a -> b
fib _ = 1
```

Haskell

Lösung

Haskell

```haskell
        module Fib where
fib :: (Integral a, Integral b) => a -> b
fib n = round ((((1+w5)/2)^n-((1-w5)/2)^n)/w5)
  where
    w5 = sqrt 5
```

Haskell

Erläuterung

Zum Glück lässt sich die Formel mehr oder weniger direkt abschreiben.
Die so berechnete Zahl muss mit `round` von einer Fließkommazahl wieder zu einer ganzen Zahl gewandelt werden. Einer der Tests hat (`fib 1000`) getestet. Probieren Sie das auch mal im Interpreter.

Frage: Logische Implikation

Definieren Sie einen Operator ->, der die logische Implikation realisiert. Haskell

```haskell
module Logic where
(-->) :: Bool -> Bool -> Bool
x --> y = False
```

Haskell

Lösung

Haskell

```haskell
module Logic where
(-->) :: Bool -> Bool -> Bool
x --> y = not x || y
```

Haskell

Erläuterung

Frage: Logik Transformation

Gegeben sei ein Datentyp `Logic` für logische Formeln mit $\wedge$, vee und $\neg$. Schreiben Sie eine Funktion, die eine Formel in eine logisch äquivalente Formel umwandelt, in der Negationen nur direkt vor Atomen stehen. Hierzu sind folgende Transformationen vorzunehmen:

- $\neg\neg A$ zu A.
- $\neg(A \vee B)$ zu $(\neg A \wedge \neg B)$.
- $\neg(A \wedge B)$ zu $(\neg A \vee \neg B)$.

Haskell

```haskell
      module Logic where
data Logic =
   Atom String
  |Logic :/\ Logic
  |Neg Logic
  |Logic :\/ Logic
    deriving (Show, Eq)

norm :: Logic -> Logic
norm x = x
```

Haskell

Lösung

Haskell

```haskell
        module Logic where
data Logic =
   Atom String
  |Logic :/\ Logic
  |Neg Logic
  |Logic :\/ Logic
    deriving (Show, Eq)

norm :: Logic -> Logic
norm (Neg (Neg a)) = norm a
norm (Neg (a :\/ b))
 = norm (Neg a) :/\ norm (Neg b)
norm (Neg (a :/\ b))
 = norm (Neg a) :\/ norm (Neg b)
norm (a :\/ b) = norm a :\/ norm b
norm (a :/\ b) = norm a :/\ norm b
norm x = x
```

Haskell

Erläuterung

Die Transformationsregeln lassen sich direkt über *Pattern Matching* definieren. Es darf nicht vergesen werden, rekursiv in den Ausdruck hinein zu laufen.

Frage: Euklid

Implementieren Sie den Euklid-Algorithmus zur Berechnung des größten gemeinsamen Teilers.

$$euklid(a, 0) = a$$

$$euklid(a, b) = euklid(b, a \ mod \ b)$$

Haskell

```haskell
module Euklid where
euklid :: Integral t => t -> t -> t
euklid _ _ = 1
```

Haskell

Lösung

Haskell

```haskell
module Euklid where

euklid :: Integral t => t -> t -> t
euklid a 0 = a
euklid a b = euklid b (a `mod` b)
```

Haskell

Erläuterung

Die Gleichungen des Algorithmus lassen sich direkt über *Pattern Matching* definieren.

Frage: Primzahl

Schreiben Sie eine Funktion, die die n-te Primzahl berechnet. Die erste Primzahl ist die 2. Definiieren Sie hierzu die Liste aller Primzahlen nach dem Verfahren des Sieb des Erathostenes. Haskell

```haskell
      module Prime where
prime :: Int -> Integer
prime n = 1
```

Haskell

Lösung

Haskell

```haskell
        module Prime where
prime :: Int -> Integer
prime n = primes !! (n-1)

primes = sieb [2..]

sieb (x:xs) = x:sieb [y|y<-xs,y `mod` x /= 0]
```

Haskell

Erläuterung

Mit einer Mengenschreibweise auf Listen lässt einfach die Liste quf alle Zahlen filtern, die nicht durch das erste Elemente teilbar sind.