(kurze) Geschichte der Programmierung

Beginn der Programmiersprachen in den 30iger Jahren. Pioniere sind Alonso Church und Stephen Kleene. Es wurden theoretische Sprachen entworfen. Wichtigster Kandidat ist das Lambda Kalkül. Wir reden hier über eine Zeit, als es noch keine Computer gab. Die Programmiersprachen waren theoretisch, und nicht praktisch auf Hardware anwendbar, da diese schlicht fehlte. In dieser Zeit hat Alan Turing sich über die Berechnung des Entscheidungsproblems Gedanken gemacht hat. Er erdachte die nach ihm benannte Turing Maschnine, ein Modell, um die Klasse der intuitiv berechenbaren Funktionen zu bilden.

In den 40iger Jahren wurden erste Programmgesteuerte Computer erfunden, Zuse ist hier ein Vorreiter. Diese Computer wurden mit Lochkarten programmiert. Das Wort Software gab es so noch nicht. Es bildete sich erst aus dem Wort Layette für Programme bei der UNIVAC Corporation.

Imperative Programmiersprachen wie sie heute bekannt sind wurden durch das Von-Neumann Prinzip im Jahr 1945 begründet. Das Konzept des „conditional-control-transfer“ beinhaltet:

- if then Anweisungen
- looped Anweisungen
- subroutines heute als Funktionen bekannt

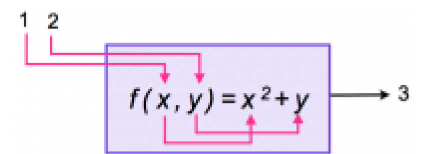
In den 50igern entwickelten sich dann die imperativen Programmiersprachen, wie wir sie heute kennen.

Haskell

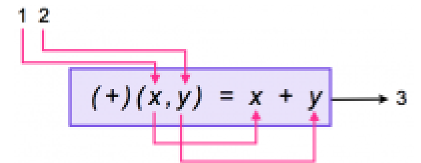
deklarativ	imperativ
Sprachen basieren auf einem mathematischen Formalismus	Sprachen sind von der dahinter stehenden Hardware geprägt
Wissen über ein Problem rein deklarativ darstellbar	Befehlssequenz (Zustände)
Intelligentes System, das Fragen an das Programm beantworten kann	zeitlicher Ablauf im Programm sichtbar
kompakter robuster einfacher	effizienter

- Rein funktional
- Keine seiteneffekte
- Referenzielle Transparenz: der Wert eines Ausdrucks ist von der Zeit seiner Ausführung unabhängig

Funktionalität



Haskell ist eine reine Funktionale Programmiersprache, mit einer nach-Bedarf-Auswertung-Strategie die man als „Lazy Evaluation“ bezeichnet. Funktional bedeutet, dass Programme als mathematische Funktionen dargestellt werden. Diese Eigenschaft fürt dazu, dass Haskell-Funktionen keine Seiteneffekte haben. Dadurch entsteht refernzielle Transparenz. Das heißt, der Wert eines Ausdrucks hängt nur von den Werten seiner Parameter ab und ist zeitlich konstant.



Funktionen können auch einfache arithmetische Operationen sein. Es ist ein Unterschied, ob man + 2 3 oder 2 + 3 schreibt. + 2 3 ist ein Aufruf der Funktion + mit den Parametern 2 und 3. Das andere ist eine einfache Addition der der beiden Zahlen.

Datentypen

Name	Beschreibung	Werte
Int	ganzzahlige Werte	$-2^{31} ... 2^{31}-1$
Integer	ganzzahlige Werte	unbeschränkt
Bool	Wahrheitswerte	`True` `False`
Char	Zeichen	'a' '1' '+'
Float	Gleitkommazahlen	32-Bit
Double	Gleitkommazahlen	64-Bit
Typ$_1$ → Typ$_2$	Funktionen	

Operationen

Symbol	Operator	Priorität
||	ODER	**8**
&&	UND	**7**
===	Gleichheit	**6**
/=	Ungleichheit (!=)	**6**
<	Kleiner	**5**
>	Größer	**5**
⇐	Kleiner Gleich	**5**
>=	Größer Gleich	**5**
+	Addition	**4**
-	Subtraktion	**4**
*****	Multiplikation	**3**
/	Division	**3**
`div`	ganzzahlige Division	**3**
`mod`	'Rest' bzw. Modulo	**3**
******	ganzzahlige Potenz	**2**
^	Potenz	**2**
not	logische Negation	**1**

Fallunterscheidung, Schleifen, Abfragen

If - then - else

```
 sign :: Int -> Int
sign x = if x > 0
         then 1
         else if x<0
              then -1
              else 0
```

Case

```
 german2italian :: Char -> String
german2italian x = case x of
                     'c' → "do"
                     'd' → "re"
                     'e' → "mi"
                     'f' → "fa"
                     'g' → "sol"
                     'a' → "la"
                     'h' → "si"
```

Guards

```
 sign :: Int -> Int
sign x
      | x>0         = 1
      | x===0        = 0
      | otherwise   = -1
```

lokale Funktionsdefinition

```
  f :: Float -> Float -> Float
f x y = (a+1)*(b+2)
        where
          a = (x+y)/2
          b = (x+y)/3
```

Begriffe

Bottom

Wenn die Auswertung eines Ausdrucks zu einer unendlichen Folge von Reduktionen führt, wird entweder das Programm nicht beendet oder es stürzt ab, weil der Speicher voll wird. In der Theorie

wird das Symbol Bottom ⊥ verwendet, um den Wert von Ausdrücken darzustellen, die nicht vollständig ausgewertet werden können (die divergent sind).

Strikte Funktionen

Informell kann man sagen, dass eine Funktion `f` strikt nach einem ihrer Argumente a ist, wenn für die Auswertung der Funktion, die Auswertung von a notwendig ist.

Normalform

das ist die form, auf die ein Ausdruck zurückgeführt werden kann.

Auswertungsstrategien

Call-by-Value

Ausdrücke werden von innen nach außen und von links nach rechts ausgewertet.

```
z=2*5
f x = x+x
f z = f(2*5);
    = f 10;
    = 10+10;
    = 20
```

Call-by-Name

Ausdrücke werden außen nach innen ausgewertet.

```
f x = f(2*5);
    = (2*5)+(2*5;
    = 10+(2*5);
    = 10+10;
    = 20
```

Call-by-Need oder Lazy Evaluation

Diese Strategie wird in Haskell verwendet - es wird lediglich das ausgewertet, was auch benötigt wird. Die Definition einer unendlichen Liste mit `[…]` ist möglich wird aber nur soweit ausgegeben/berechnet, wie es für eine Berechnung notwendig ist. Haskell selbst ist somit nicht strikt, denn Ausdrücke werden nur bei Bedarf ausgewertet.

```
z*z where z=2*5;
z*z where z=10;
```

```
            10*10;
            100
```

Kommentare

```
{-
  .......Blockkommentare ......
  ....................................
 -}

-- Zeilenkommentare
```

Pattern Matching

Durch pattern matching ist es in Haskell möglich auf Stellen von Variablen zuzugreifen. So ist die erste Stelle einer Liste mit `x:xs` ansprechbar. Wenn für eine Variable es für eine Variabel egal ist welchen Wert sie hat, kann man das mit einem Unterstrich _ zeigen.

```
 und :: Bool -> Bool -> Bool
und True True = True
und _    _    = False
```

Type Synonyme

Mit Typ-Synonymen kann man die Lesbarkeit von Programmen durch die Nutzung von Daten Tupeln erhöhen.

```
 type Point = (Double, Double)

distance :: Point -> Point -> Double
distance (x1,y1) (x2,y2) = sqrt (sumSq (x1-x2) (y1-y2))
                           where
                              sumSq x y = x*x + y*
```

Rekursion

Rekursion ist ein fundamentales Konzept in der funktionalen Programmierung. Sie steht in einer engen Beziehung zur mathematischen Induktion. Für eine Rekursion benötigt man einen Rekursionsanker, das ist eine feste Zuweisung eines bestimmten Rekursionsschrittes zu einem eindeutigem Wert. Der Rekursionsanker wird in der Regel für den kleinsten möglichen Wert definiert, mit dem die Funktion aufgerufen werden kann.

Die Funktion wird meistens mit dem größten Wert n aufgerufen um danach mit sich selber und `n-1` aufgerufen zu werden.

```
-- Wie viele Permutationen kann eine Reihe von Objekten maximal haben?
 factorial :: Int -> Int
factorial n
            | n===0 = 1
            | otherwise = n*factorial (n-1)

-- Größter gemeinsamer Teiler von zwei natürlichen Zahlen a, b nach Euklid
(originaler Algorithmus)
 ggt :: Integer → Integer → Integer
ggt pq
      |p>q   = ggt(p-q)q
      | p===q = p
      | p<q  = ggt p (q-p)

-- eine andere Variante des ggt mit mod
 ggt a b
       | a===0 && b===0 = error "ggt00notdefined"
       | b === 0       = a
       | otherwise    = ggt b (a `mod` b)
```

Listen

Listen sind die wichtigsten Datenstrukturen in funktionalen Programmiersprachen sie stellen Sammlungen von Objekten dar, die den gleichen Datentyp besitzen. Sie sind dynamische Datenstrukturen.

Listen sind rekursive Strukturen: Eine Liste ist entweder leer `[]` oder ein konstruierter Wert, der aus einem Listenkopf x und einer Restliste `xs` besteht. Der Typ einer Liste, die Elemente des Typs `t` enthält, wird mit `[t]` bezeichnet. Beispiele für verschiedene Listen:

```
[1,2,3]           :: Integer
1:[0,3,7]         :: Integer
[[0.3, 0.0], []] :: [[Double]]
[(3,0), (2,3)]   :: [(Integer,Integer)]

 kopf :: [ Integer ] -> Integer
kopf (x:xs) = x
kopf [] = error "ERROR: empty list"

rumpf :: [ Integer ] -> [ Integer ]
rumpf (x:xs) = xs
rumpf [] = error "ERROR: empty list"

 summe :: [Integer] -> Integer
summe ls = if ls == []
           then 0
```

```
           else ( kopf ls ) + summe (rumpf ls)

-- mit Pattern Matching
 summe :: [Integer] -> Integer
summe [] = 0
summe (x:xs) = x + summe xs

multList :: [Integer] -> Integer
multList [] = error "the function is not defined for []" multList [x] = x
multList (x:xs) = x * multList xs

laenge :: [Int] -> Int
laenge [] = 0
laenge (x:xs) = 1 + laenge xs

(++) :: [Int]->[Int]->[Int] (++)[]ys = ys
(++) (x:xs) ys = x:(xs ++ ys)

[5,6] ++ [1,2,3] ⇒ 5: ([6] ++ [1, 2, 3])
                 ⇒ 5: (6: ([] ++ [1, 2, 3]))
                 ⇒ 5: (6: [1, 2, 3])
                 ⇒ 5: [6, 1, 2, 3]
                 ⇒ [5, 6, 1, 2, 3]
```

Anwendungen

Fibonacci

Der Fibonacci Algorithmus ist beschreibt eine Zahlenfolge, die man in der Natur sehr häufig wiederfinden. So zum Beispiel in der Anzahl der Äste eine Baumes. Der Algorithmus ist sehr simpel

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Allerdings berechnet die `fib` Funktion wie sie hier gezeigt ist zu oft den selben Wert, so ruft `fib40` 331 160 281 die `fib` Funktion auf. Die Anzahl der Reduktionen von `fib n` ist `fib n`+1. Eine Lösung diese Problems ist, die ersten beiden Zahlen der `fib` Funktion zu übergeben und jedes mal die Funktion mit `n-1` aufzurufen.

```
fib1 n = quickFib 0 1 n
         where
            quickFib a b 0 = a
           quickFib a b n = quickFib b (a+b) (n-1)
```

Die `quickFib` Funktion funktioniert nur, wenn diese mit den ersten zwei Zahlen der Fibonacci Reihe aufgerufen wird. n ist der Zähler, um zu sehen, das wievielte mal `quickFib` aufgerufen wurde. Innerhalb jedes rekursiven Aufrufs wird eine neue Fibonacci-Zahl berechnet und der Zähler verkleinert. Die neue Zahl und ihr Vorgänger werden beim nächsten rekursiven Aufruf als Parameter weitergegeben. Für die Berechnung von `quickFib n` benötigen wir genau n Reduktionen.

Eine weitere Möglichkeit ist, die Lösung mit Tupeln:

```
 nextFib :: (Integer, Integer) -> (Integer, Integer)
nextFib (a,b) = (b, a+b)

fib n = fst ( fibTuple n )

fibTuple n
           | n===0 = (0, 1)
           | otherwise = nextFib (fibTuple (n-1))
```

Die Fibonacci Zahlen stehen in einem Engen Verhältnis zum Goldenen Schnitt, bei dem zwei positive reelle Zahlen wie folgt im Verhältnis stehen:

$\frac{a}{b} = \frac{a+b}{a}$ für $a > b > 0$.

Dieses Verhältnis lässt sich durch folgende Beziehung ausdrücken:

$$fib(n) = \frac{1}{\sqrt{5}} \left(\left(1 + \frac{\sqrt{5}}{2} \right)^2 - \left(1 - \frac{\sqrt{5}}{2} \right)^2 \right)$$

Diese Formel kann man 1:1 in Haskell Code umsetzen:

```
 aproxFib :: Double -> Double
aproxFib n = (1/(sqrt5))*(a**n - b**n)
             where
               a = (1 + sqrt5)/2
               b = (1 - sqrt5)/2
               sqrt5 = sqrt 5
```

Binärzahlen

Zahlen sind immer nur in Ihrem jeweiligen Zahlensystem gültig. So ist eine Binärzahl im Binärsystem gültig, eine Dezimalzahl im Dezimalsystem. Um zu zeigen, dass eine Zahl zu einem System gehört, wird sie mit einem Indize geschrieben, der die Anzahl an Zeichen angibt, die in dem jeweiligem Zahlensystem vorhanden sind. So hat das Binäre Zahlensystem 2 Stellen nämlich {0,1} und das Dezimalsystem 10 Zeichen {0,1,2,3,4,5,6,7,8,9}. Um von einem Systemen ein anderes umzurechnen

muss Jede Stelle der Zahl mit der Anzahl an Zeichen hoch dem Index der Stelle genommen werden.

$$Zahl_{AnzahlZeichenQuellsystem} = ZeichenAnStelle \times AnzahlZeichen^{StelleInZahl}$$

$$101010_2 = 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$$
$$101010_2 = 32 + 8 + 2$$
$$101010_2 = 42_{10}$$

Allgemein ausgedrückt: $z_n z_{n-1} z_{n-2} \cdots z_2 z_1 z_0 = \sum_{i=0}^{n} z_i \times b^i$ wobei z das Zeichen an der Stelle i ist mit der Basis b.

Vereinfacht lässt sich sagen, dass man um die 42_{10} in das Binärsystem umzurechnen einfach so lange durch 2 Teilen muss, bis 0 rauskommt. Die einzelnen Zwischen Ergebnisse kann man mittels Modulo auf Reste untersuchen und das Ergebnis ergibt dann rückwärts gelesen die Binäre Zahl.

div2	mod2	Leserichtung
42	0	
21	1	^
10	0	|
5	1	|
2	0	|
1	1	|
0	0	

Um eine Umrechnung vom Dezimalsystem zum Binärsystem mit Haskell zu realisieren, muss man sich ansehen, wie man die eben gerade gezeigte Art zu rechnen mit den Methoden in Haskell darstellen kann.

```
dec2bin :: Int -> [Int]
dec2bin n
          | n<2 = [n]
          | otherwise = dec2bin (n`div`2) ++ [n `mod`2]
```

Eine weitere Möglichkeit der Umwandlung von Binären in dezimale Zahlen ist, die 2 aus den Potenzen nach vorne zu ziehen, somit wird die Berechnung überschaubarer, rekursiv und Programmnäher.

$$101101_2 = 1\times2^5+0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0$$

$$101101_2 = 2\left(1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0\right)+1$$

$$101101_2 = 2\left(2\left(1\times2^3+0\times2^2+1\times2^1+1\times2^0\right)+0\right)+1$$

$$101101_2 = 2\left(2\left(2\left(1\times2^2+0\times2^1+1\times2^0\right)+1\right)+0\right)+1$$

$$101101_2 = 2\left(2\left(2\left(2\left(1\times2^1+0\times2^0\right)+1\right)+1\right)+0\right)+1$$

$$101101_2 = 2\left(2\left(2\left(2\left(2\left(1\times2^0\right)+0\right)+1\right)+1\right)+0\right)+1$$

$$101101_2 = 2\left(2\left(2\left(2\left(2\right)+1\right)+1\right)+0\right)+1$$

$$101101_2 = 2\left(2\left(2\left(5\right)+1\right)+0\right)+1$$

$$101101_2 = 2\left(2\left(11\right)+0\right)+1$$

$$101101_2 = 2\left(22\right)+1$$

$$101101_2 = 45$$

```
drehe [] = []
drehe (x:xs) = drehe xs ++ [x]
```

```
binary2decimal xs = bin2dec ( drehe xs)

bin2dec :: [Int] -> Int
bin2dec [] = 0
bin2dec (x:xs) = 2*(bin2dec xs) + x
```

Addition

Binärzahlen kann man addieren, diese erfolgt nach dem einfachem Prinzip, dass $0+1=1, 0+0=0, 1+1=0$ (Rest 1).

Übertrag	_1_11__11__
Zahl 1	_1011001101
Zahl 2	_1011000110
Summe	10100010011

Eine Möglichkeit dies in Haskell zu implementieren ist:

```
binaryAdd :: [Int] -> [Int] -> [Int]
binaryAdd b1 b2 = reverse (binarySum 0 (reverse b1) (reverse b2))
                  where
                        binarySum u [] ys = ys ++ [u]
                        binarySum u xs [] = xs ++ [u]
                        binarySum 0 (0:xs) (0:ys) = 0: (binarySum 0 xs ys)
                        binarySum 0 (0:xs) (1:ys) = 1: (binarySum 0 xs ys)
                        binarySum 0 (1:xs) (0:ys) = 1: (binarySum 0 xs ys)
                        binarySum 0 (1:xs) (1:ys) = 0: (binarySum 1 xs ys)
                        binarySum 1 (0:xs) (0:ys) = 1: (binarySum 0 xs ys)
                        binarySum 1 (0:xs) (1:ys) = 0: (binarySum 1 xs ys)
                        binarySum 1 (1:xs) (0:ys) = 0: (binarySum 1 xs ys)
                        binarySum 1 (1:xs) (1:ys) = 1: (binarySum 1 xs ys)
```

Funktionen mit Listen

```
exoder :: Int -> Int -> Int
exoderxy | x==y =0
         | otherwise = 1

bitXOr :: [Int] -> [Int] -> [Int]
bitXOr []      []      = []
bitXOr (x:xs) (y:ys) = (exoder x y): (bitXOr xs ys)
bitXOr _       _       = error "the two lists are not of the same length"

balanced:: [Char ] -> Bool
balanced ls = bal [] ls
```

```
bal:: [Char] -> [Char] -> Bool
bal [] [] = True
bal s ('(':xs)  = bal ('(':s) xs
bal s ('[':xs)  = bal ('[':s) xs
bal s ('{':xs)  = bal ('{':s) xs
bal ('(':xs) (')':ys) = bal xs ys
bal ('[':xs) (']':ys) = bal xs ys
bal ('{':xs) ('}':ys) = bal xs ys
bal _ _ = False
```

Zettel

Übung 1

```
{-==============================8<============================================
========================  Aufgabe 1 ==========================================

Wenn die Seitenla ̈ngen eines rechtwinkligen Dreiecks natu ̈rliche Zahlen
sind,
werden diese Zahlen als pythagora ̈ische Zahlentripel bezeichnet. Definieren
Sie eine Funktion in Haskell, die bei Eingabe dreier natu ̈rlicher Zahlen
feststellen
kann, ob es sich um die Seitenla ̈ngen eines rechtwinkligen Dreiecks handelt
oder nicht.
Mit anderen Worten soll die Funktion testen, ob die eingegebenen natu
̈rlichen Zahlen
pythagora ̈ische Zahlentripel sind.

==============================8<==========================================-}

--multiplies a number with itsself
--quad :: Int -> Int -> Int
quad x = x*x

--check three numbers if they are a 'pythagorisches tripel'
--checktripel :: Int -> Int -> Int -> Bool
checktripel a b c
            | c == sqrt(quad(a)+quad(b)) = True
            | otherwise          = False

--Die Typdeklaration verursacht Fehlermeldungen beim kompilieren,
--ohne geht es aber - wie kann man das besser machen? ich will doch Integer
eingeben und nen Bool bekommen.
```

```
{-=============================8<=========================================
========================  Aufgabe 2 =====================================

Schreiben Sie ein Funktion, die bei Eingabe eines Jahres entscheidet, ob es
sich um einem
Schaltjahr handelt.

==============================8<========================================-}

--check if x is multiple of y
--chkModNull::Int->Int->Bool
chkModNull x y
            | mod x y == 0  = True
            | otherwise     = False

--check if x is switchyear
--chkSwitchYear::Int->Bool
chkSwitchYear x = ((chkModNull x 4) && not(chkModNull x 100)) || (chkModNull
x 400)

{-=============================8<=========================================
========================  Aufgabe 3 =====================================

Schreiben Sie eine Funktion, die bei Eingabe von drei Zahlen a, b und c
entscheiden kann, ob
eine der drei Zahlen ein Mehrfaches der anderen zwei Zahlen ist.

==============================8<========================================-}

--check if x is multiple of y
--chkModNull::Int->Int->Bool
chkModNull x y
                | mod x y == 0  = True
                | otherwise     = False

--get the number wich is less then the other
getLittle::Int->Int->Int
getLittle x y
                | x>y = y
                | otherwise = x

--get the number wich is more then the other
getTall::Int->Int->Int
getTall x y
                | x>y = x
                | otherwise = y

isDivisor x y z
                | (chkModNull (getTall x y) (getLittle x y)) && (chkModNull
(getTall x z) (getLittle x z)) == True = True
                | otherwise = False
```

```
--check if x is multiple of a and b
--chkMultiple::Int->Int->Int->Bool
chkMultiple a b c
                    | isDivisor(a b c) && isDivisor(b a c) && isDivisor(c a
b) == True = True
                    | otherwise = False

{-============================8<==========================================
=========================  Aufgabe 4 ======================================

Schreiben Sie ein Funktion toLower, die beliebige Buchstaben in
Kleinbuchstaben umwandelt.

==============================8<========================================-}

--convert a given Upper Case Char to a lower Case Char by getting the
Dezimal
--Value and adding 32 to get the right ASCII Dezimal Value
--toLower::Char->Char
toLower x = toEnum(fromEnum x +32)::Char

{-============================8<==========================================
=========================  Aufgabe 5 ======================================

Schreiben Sie eine Funktion, die ein Zeichen als Argument bekommt und
entscheiden kann,
ob das Zeichen ein Buchstabe ist.

==============================8<========================================-}

--checks if a given Char is a Letter or not
--toLower::Char->Bool
isLetter x
            | fromEnum x > 64 && fromEnum x < 91     = True --one more and
less to get a >=
            | fromEnum x > 96 && fromEnum x < 123    = True
            | otherwise = False

{-============================8<==========================================
=========================  Aufgabe 6 ======================================

Definieren Sie eine Funktion weekday in Haskell, die bei Eingabe eines
Datums
(in Form von drei positiven Zahlen) den Wochentag-Namen mit Hilfe folgender
Formeln des Gregorianischen Kalenders berechnet. Die Formeln berechnen eine
Zahl zwischen 0 (Sonntag) und 6 (Samstag).

==============================8<========================================-}

f :: Int -> Float
```

```
f month = (14- month)/(12)

y :: Int -> Int -> Float
y year month = year - (f month)

x :: Int -> Int -> Float
x year month = (y year month) + ((y year month)/4) - ((y year month)/100) +
((y year month)/400)

m :: Int -> Float
m month = month + 12 * (f month) -2

name :: Int -> Int -> Int -> Int
name year month day = mod (round (day+(x year month)+((31*(m month))/(12))))
7

weekday :: Int -> Int -> Int -> String
weekday year month day
                    | name year month day == 1 = "Montag"
                    | name year month day == 2 = "Dienstag"
                    | name year month day == 3 = "Mittwoch"
                    | name year month day == 4 = "Donnerstag"
                    | name year month day == 5 = "Freitag"
                    | name year month day == 6 = "Samstag"
                    | name year month day == 7 = "Sonntag"

{-=============================8<=======================================
========================  Aufgabe 7 =======================================

Die Fla ̈che eines beliebigen regula ̈ren Polygons kann bei Eingabe der
Seitenla ̈ngen s und der Anzahl der Seiten n mit Hilfe folgender Formeln
berechnet werden. Schreiben Sie entsprechende Haskell - Funktion, die die
Berechnung macht. Verwenden Sie lokale Funktionen (mit Hilfe der
where-Anweisung)
fu ̈r die Berechnung der Teilformeln.

==============================8<======================================-}

--area :: Double -> Double -> Double
area n s = (n*s*apothema)/2
        where
            --apothema :: Double -> Double -> Double
            apothema n s = (s/(2*tan (pi/n)))
```

Übung 2

```
{-=============================8<=======================================
========================  Aufgabe 1 =======================================
```

```
Nehmen Sie an, wir haben folgende Datentyp-Synonyme definiert:
type Point = (Double, Double) type Rectangle = (Point, Point)
Definiere unter Verwendung des Rectangle-Datentyps folgende Funktionen:
area :: Rectangle -> Double
overlaps :: Rectangle -> Rectangle -> Bool -- Testet, ob die Rechtecke sich
überlappen contains :: Rectangle -> Rectangle -> Bool -- Testet, ob eines
der Rechtecke
das andere Rechteck beinhaltet

==============================8<=========================================-}

-- Typdefinitionen
type Point = (Double, Double)
type Rectangle = (Point, Point)
-- Die Kanten dieses Rechtecks sind parallel zu X- bzw. Y-Achse.
--  Das Rechteck ist das kleinstmoegliche, das seine definierenden
--  Punkte enthaelt (d.h. die definierenden Punkte sind diagonal
--  gegenueberliegende Ecken des Rechtecks )

-- Hilfsfunktionen

-- Minimaler X-Wert der Punkte im Rechteck
minX :: Rectangle -> Double
minX ( (x1, y1), (x2, y2) ) = min x1 x2

-- Maximaler X-Wert der Punkte im Rechteck
maxX :: Rectangle -> Double
maxX ( (x1, y1), (x2, y2) ) = max x1 x2

-- Minimaler Y-Wert der Punkte im Rechteck
minY :: Rectangle -> Double
minY ( (x1, y1), (x2, y2) ) = min y1 y2

-- Maximaler Y-Wert der Punkte im Rechteck
maxY :: Rectangle -> Double
maxY ( (x1, y1), (x2, y2) ) = max y1 y2

-- 1a) Flaeche eines Rechtecks
area :: Rectangle -> Double
area ( (x1, y1), (x2, y2) ) = abs( (x1-x2) * (y1-y2) )

-- 1b) Ueberlappen von zwei Rechtecken
overlaps :: Rectangle -> Rectangle -> Bool

overlaps r1 r2 =
    -- Teste, ob die Ausdehnung der X- UND die der Y-Dimension der
    --  Rechtecke ueberlappt.
    --  Dafuer muss einer der minimalen X- bzw. Y-Werte der Rechtecke
    --  zwischen dem minimalen und maximalen X- bzw. Y-Wert des anderen
    --  liegen.
```

```
    ( ( minX r2 <= minX r1 && minX r1 <= maxX r2 ) || ( minX r1 <= minX r2
&& minX r2 <= maxX r1 ) )
    && ( ( minY r2 <= minY r1 && minY r1 <= maxY r2 ) || ( minY r1 <= minY
r2 && minY r2 <= maxY r1 ) )

-- 1c) Beinhaltet ein Rechteck alle Punkte des anderen?
contains :: Rectangle -> Rectangle -> Bool

contains r1 r2 =
    -- Aehnlich wie oben, nur dass die Ausdehnung eines der Rechtecke in
    --  beiden Dimensionen in der des anderen Rechtecks enthalten sein
    --  muss.
    ( ( minX r2 <= minX r1 ) && ( maxX r1 <= maxX r2 )
           && ( minY r2 <= minY r1 ) && ( maxY r1 <= maxY r2 ) )
    || ( ( minX r1 <= minX r2 ) && ( maxX r2 <= maxX r1 )
           && ( minY r1 <= minY r2 ) && ( maxY r2 <= maxY r1 ) )

{-============================8<==========================================
========================  Aufgabe 2 =======================================

a) Schreiben Sie eine rekursive Funktion, die alle ungerade Zahlen zwischen
1 und n
aufsummiert.
b) Gibt es neben der rekursiven Berechnung noch weitere
Berechnungsmöglichkeiten?
Wenn ja, programmieren Sie auch diese Funktion.

==============================8<=========================================-}
-- 2a) Summiere alle ungeraden Zahlen zwischen 1 und n auf
oddSum :: Integer -> Integer
oddSum n
    | n < 1 = error "Expecting natural number"

oddSum 1 = 1

oddSum n =
    if ( mod n 2 ) == 0
    then oddSum ( n - 1 )
    else n + oddSum ( n - 2 )

-- 2b) Alternative Methode
oddSum2 :: Integer -> Integer
oddSum2 n
    | n < 1 = error "Expecting natural number"

-- Durch Rundung von div wird 1 abgezogen, wenn n gerade ist:
--  n selbst wird als gerade Zahl ignoriert.
oddSum2 n = ( div ( n + 1 ) 2 ) ^ 2

-- Test
testOddSum2 :: Integer -> Bool
```

```
testOddSum2 n
    | n < 1 = error "Expecting natural number"

testOddSum2 1 = oddSum 1 == oddSum2 1
testOddSum2 n =
    if oddSum n == oddSum2 n
    then testOddSum2 ( n - 1 )
    else False

{-==============================8<===========================================
========================  Aufgabe 3 ==========================================

Innerhalb eines Rechners werden Wahrheitswerte nur mit Hilfe der Binärzahlen
1 und 0 dargestellt. Nehmen wir an, wir haben folgende Haskell-Funktionen
definiert:
true :: Int true = 1
false :: Int false = 0
Definieren Sie die logischen Funktionen oder, exoder, und und negation nur
mit
Hilfe von arithmetischen Operationen.

==============================8<==========================================-}

-- Definitionen

true :: Int
true = 1

false :: Int
false = 0

-- 3a)
oder :: Int -> Int -> Int
oder a b = div ( a + b + 1 ) 2

-- 3b)
exoder :: Int -> Int -> Int
exoder a b = ( a - b ) * ( a - b )

-- 3c)
und :: Int -> Int -> Int
und a b = div ( a + b ) 2

-- 3d)
negation :: Int -> Int
negation a = 1 - a

{-==============================8<===========================================
========================  Aufgabe 4 ==========================================

Schreiben Sie eine Funktion, die bei Eingabe einer positiven Zahl die Einsen
```

```
der Binärstellung der Zahl addiert (Quersumme der Binärdarstellung der
Zahl).

==============================8<=========================================-}
binSum :: Int -> Int
binSum n
    | n < 0 = error "Expecting positive number"
binSum 0 = 0
binSum n = ( binSum ( div n 2 ) ) + ( mod n 2 )

{-=============================8<==========================================
========================  Aufgabe 5 =======================================

Definieren Sie eine Haskell-Funktion, die bei Eingabe einer Zahl in
Hexadezimal-
Darstellung die Oktal-Darstellung der Zahl berechnet.
Die Zahl soll als Zeichenkette eingegeben werden.
Anwendungsbeispiel: hex2okt "1F8" => "0770"

==============================8<=========================================-}
hex2oct :: String -> String
hex2oct "" = error "Empty Number"
hex2oct s = rbin2oct ( num2rbin ( hex2num s ) )

hex2num :: String -> Integer
hex2num "" = error "Empty Number"
hex2num s = hex2numSub 0 s

hex2numSub :: Integer -> String -> Integer
hex2numSub x [] = x
hex2numSub x (c:cs) = hex2numSub ( 16 * x + ( char2num c ) ) cs

num2rbin :: Integer -> String
num2rbin 0 = "0"
num2rbin n = ( if mod n 2 == 1 then "1" else "0" ) ++ ( num2rbin ( div n 2 ) )

rbin2oct :: String -> String

rbin2oct (c1:c2:c3:cs) =
    rbin2oct( cs ) ++
    [num2char( char2num( c1 ) + char2num( c2 ) * 2 + char2num( c3 ) * 4 )]
rbin2oct (c1:c2:[]) = [num2char( char2num( c1 ) + char2num( c2 ) * 2 )]
rbin2oct (c1:[]) = [num2char( char2num( c1 ) )]
rbin2oct [] = ""

-- Wandelt eine Ziffer in hexadezimaler Darstellung (was binaere, oktale und
dezimale
--  Darstellung einschliesst) in einen Integer um
char2num :: Char -> Integer
char2num c
    | c >= '0' && c <= '9' = fromIntegral ( ( fromEnum c ) - ( fromEnum '0' ) )
```

```
    | c >= 'A' && c <= 'F' = 10 + fromIntegral ( ( fromEnum c ) - ( fromEnum
'A' ) )
    | otherwise = error ( "Unknown digit char: " ++ [c] )

-- Wandelt einen Integer in eine Ziffer in hexadezimaler Darstellung um
num2char :: Integer -> Char
num2char n
    | n >= 0 && n < 10 = toEnum ( ( fromEnum '0' ) + ( fromIntegral n ) )
    | n >= 10 && n < 16 = toEnum ( ( fromEnum 'A' ) + ( fromIntegral ( n -
10 ) ) )
    | otherwise = error "No known char for number"

{-============================8<=========================================
========================  Aufgabe 6 =======================================

Definieren Sie eine Funktion digitSum, die die Quersumme einer Zahl so lange
berechnet,
bis das Ergebnis nur aus einer Ziffer besteht (Zahl zwischen 0-9)
Anwendungsbeispiel: digitSum 352418 => 5

==============================8<=========================================-}
digitSum :: Integer -> Integer
digitSum n =
    if a == n
    then a
    else digitSum a
    where a = simpleDS n

-- Helferfunktion einfache Quersumme
simpleDS :: Integer -> Integer
simpleDS 0 = 0
simpleDS n = ( simpleDS ( div n 10 ) ) + ( mod n 10 )

{-============================8<=========================================
========================  Aufgabe 7 =======================================

Programmieren Sie eine Funktion, die die Elemente aus zwei Listen addiert.
Verwenden Sie die error-Funktion für den Fall, dass die Listen nicht gleich
lang sind.
Anwendungsbeispiel:
addLists [2,4,0,1] [0,1,0,2] => [2,5,0,3]

==============================8<=========================================-}
addLists :: [Integer] -> [Integer] -> [Integer]

addLists (x:xs) (y:ys) = (x+y) : ( addLists xs ys )
addLists [] [] = []
-- Eine der beiden Listen hat keine Elemente mehr
addLists _ _ = error "Lists must have the same number of elements"
```

Übung 3

```
-- Typdefinition Binaerzahl
type Bin = [Int]

-- Anmerkung: Die Binaerzahl [] ist im folgenden valide und wird
--  wie [0] behandelt, abgesehen davon, dass ihre Laenge 0 ist
--  (siehe dazu z.B. binNot und rbinSuccCut)

-------- 0 Allgemeine Hilfsfunktionen

-- Ermittelt ob die gegebene Liste den gegebenen Wert enthaelt
contains :: Eq a => [a] -> a -> Bool -- a ist ein vergleichbarer Typ
contains [] x = False
contains (x:xs) y = if( x == y ) then True else contains xs y

{-=============================8<=============================================
========================  Aufgabe 1 ==========================================

Definieren Sie eine Haskell-Funktion, die aus einer beliebigen Binärzahl n
(Zweierkomplement- Darstellung) die entsprechende negative Zahl (-n)
berechnet.
Es wird selbstverständlich angenommen, dass die Eingabe- und Ergebniszahl
der
Funktion immer die gleiche Bitlänge haben.
Anwendungsbeispiel: twoComplement [0,0,0,1,1,0,1,0] => [1,1,1,0,0,1,1,0]

==============================8<==========================================-}

twoComplement :: Bin -> Bin
twoComplement xs = reverse ( rbinSuccCut ( reverse ( binNot xs ) ) )

-- Hilfsfunktion: Negiert einzelne Binaerziffer
simpleBinNot :: Int -> Int
simpleBinNot 1 = 0
simpleBinNot 0 = 1
simpleBinNot _ = error ( "Not a binary digit" )

-- Hilfsfunktion: Negiert alle Ziffern einer Binaerzahl
binNot :: Bin -> Bin
binNot [] = []
binNot (x:xs) = ( simpleBinNot x ) : binNot xs

-- Nachfolger einer rueckwaerts geschriebenen Binaerzahl
--  Arbeitet mit Uebertrag, allerdings nicht an heochster Stelle,
--  sodass die Laenge der Zahl gleich bleibt.
rbinSuccCut :: Bin -> Bin
rbinSuccCut [] = []
rbinSuccCut (1:xs) = 0 : ( rbinSuccCut xs )
rbinSuccCut (0:xs) = 1 : xs
```

```
{-=============================8<==========================================
========================  Aufgabe 2 ======================================

Definieren Sie eine Funktion, die zwei positive Binärzahlen (jedes Element
ist 0 oder 1)
mit beliebiger Bitlänge multipliziert.
Anwendungsbeispiel: multiply [0,1,1,0,1,0] [1,1,1,0] => [1,0,1,1,0,1,1,0,0]

==============================8<=========================================-}

multiply :: Bin -> Bin -> Bin
multiply xs ys = reverse ( rbinMult ( reverse xs ) ( reverse ys ) )

-- Hilfsfunktion: Multipliziert zwei rueckwaerts geschriebene Binaerzahlen
rbinMult :: Bin -> Bin -> Bin
rbinMult (x:xs) (y:ys)
    | ( x /= 0 && x /= 1 ) = error "Not a binary digit"
    | ( y /= 0 && y /= 1 ) = error "Not a binary digit"

rbinMult [] _ = []
rbinMult _ [] = [] -- Performance
rbinMult (x:xs) ys = rbinAdd ( if x == 1 then ys else [0] ) ( rbinMult xs (0
:ys) )

-- Hilfsfunktion: Addiert zwei rueckwaerts geschriebene Binaerzahlen
rbinAdd :: Bin -> Bin -> Bin
rbinAdd xs ys = rbinAdd2 xs ys 0

-- Hilfsfunktion fuer rbinAdd mit Uebertrag als Argument
rbinAdd2 :: Bin -> Bin -> Int -> Bin
rbinAdd2 _ _ co
    | ( co > 1 ) = error "Carryover greater than 1"
rbinAdd2 (x:xs) _ _
    | ( x /= 0 && x /= 1 ) = error "Not a binary digit"
rbinAdd2 _ (x:xs) _
    | ( x /= 0 && x /= 1 ) = error "Not a binary digit"

rbinAdd2 [] [] 0 = []
rbinAdd2 [] [] co = [co]
rbinAdd2 [] ys co = rbinAdd2 ys [] co -- Vertausche Argumente: weniger zu
schreiben
-- ( mod sum 2 ) ist das Ergebnis, ( div sum 2 ) der Uebertrag
rbinAdd2 (x:xs) [] co = ( mod sum 2 ) : ( rbinAdd2 xs [] ( div sum 2 ) )
    where sum = x + co
rbinAdd2 (x:xs) (y:ys) co = ( mod sum 2 ) : ( rbinAdd2 xs ys ( div sum 2 ) )
    where sum = x + y + co

{-=============================8<==========================================
========================  Aufgabe 3 ======================================
```

```
Verändern Sie die balance-Funktion aus den Vorlesungsfolien, sodass in einer
beliebigen Zeichenkette kontrolliert wird, ob die Klammersetzung korrekt ist.
Anwendungsbeispiele:
    balanced "(a+b)*[x-y]/{(x+1)*5}" => True
    balanced "(a+b)*x-y)/{(x+1)*5}" => False

==============================8<========================================-}

-- Modifizierte Funktion aus der Vorlesung
balanced :: [Char] -> Bool
balanced text = bal [] text
    where
    bal :: [Char] -> [Char] -> Bool
    bal [] [] = True
    bal stapel ('(':xs) = bal (')':stapel) xs
    bal stapel ('[':xs) = bal (']':stapel) xs
    bal stapel ('{':xs) = bal ('}':stapel) xs
    -- Hinzugefuegte Zeile: Ignoriere andere Zeichen
    bal stapel (x:xs) | ( x /= ')' && x /= ']' && x /= '}' ) = bal stapel xs
    bal (s:stapel) (x:xs) | s==x = bal stapel xs
    bal _ _ = False

{-============================8<=========================================
========================  Aufgabe 4 =======================================

Definieren Sie eine polymorphe Haskell-Funktion updateList, die eine Liste
xs,
eine Position i der Liste und ein neues Element elem bekommt. Die Funktion
soll
dann eine neue Liste erstellen, in der das i-te Element der ursprünglichen
Listen mit dem neuen Element ersetzt wurde.
Anwendungsbeispiele:
updateList [1,0,1,5,6,7] 3 100 => [1,0,1,100,6,7]

==============================8<========================================-}

updateList :: [a] -> Int -> a -> [a]
updateList _ i _
    | ( i < 0 ) = error "Index out of bounds" -- Noetig, falls Liste
unendlich
updateList (x:xs) 0 y = y : xs
updateList (x:xs) i y = x : updateList xs ( i-1 ) y
updateList _ i _ = error "Index out of bounds"

{-============================8<=========================================
========================  Aufgabe 5 =======================================

In der Vorlesung wurde die randList-Funktion besprochen (siehe
Vorlesungsfolien),
die in der Lage ist bei Eingabe einer positiven Zahl n eine Liste mit n
Pseudo-
```

```
Zufallszahlen zu erzeugen. Schreiben Sie eine Funktion randUntilRepeat, die
mit
Hilfe der gleichen random-Funktion aus der Vorlesung bei Eingabe eines
Startwertes
(seed) die Liste aller Pseudo-Zufallszahlen, bis eine Wiederholung vorkommt,
berechnet.

==============================8<==========================================-}

-- (Siehe Funktion contains ganz oben)

-- Modifizierte Funktion aus der Vorlesung: statt die Laenge der Liste
--  wird jetzt das Vorhandensein einer neuen Zahl in der Liste geprueft.
randUntilRepeat :: Int -> [Int]
randUntilRepeat seed = randUntilRepeat2 [random seed]
    where
    randUntilRepeat2 xs
        | ( contains xs a ) = xs
        | otherwise = randUntilRepeat2 (a:xs)
        where
        a = random (head xs)
--    randUntilRepeat2 n xs
--        |[y|y<-xs,y==(random(head xs))] /=[]=xs
--        |otherwise = randUntilRepeat2 n ((random(head xs)):xs)

-- Helferfunktion aus der Vorlesung
random :: Int -> Int
random seed = mod ( mod ( 25173 * seed + 13849 ) 65536 ) 10000

{-============================8<==========================================
========================  Aufgabe 6 =======================================

Schreiben Sie eine Haskell-Funktion, die unter Verwendung von
Listen-Generatoren alle
Klammern aus einem beliebigen Text filtert.
onlyParenthesis "(2+4*(5-2)*[0])" => "(()[])"

==============================8<==========================================-}
-- (Siehe Funktion contains ganz oben)

onlyParenthesis :: String -> String
onlyParenthesis cs = [ c | c <- cs, contains csParenthesis c ]

-- Konstante: Liste aus allen Klammerzeichen
csParenthesis :: [Char]
csParenthesis = ['(', ')', '[', ']', '{', '}' ]

{-============================8<==========================================
========================  Aufgabe 7 =======================================

Schreiben Sie unter Verwendung von Listen-Generatoren eine encode-Funktion,
```

```
die mit Hilfe von Listen-Generatoren einen beliebigen Text nach dem
einfachen
Cesar-Verfahren verschlüsselt und eine entsprechende decode-Funktion, die
einen
verschlüsselten Text wieder entschlüsselt.

==============================8<=========================================-}

encode :: String -> Int -> String
encode cs n = [ caesarChar c n | c <- cs ]

decode :: String -> Int -> String
decode cs n = [ caesarChar c (-n) | c <- cs ]

-- Hilfsfunktion zur Konvertierung eines einzelnen Buchstabens
caesarChar :: Char -> Int -> Char
caesarChar c n
    | isULC c = toEnum ( ( fromEnum 'A' ) + ( mod ( ( lcOrd c ) + n ) 26 ) )
    | isLLC c = toEnum ( ( fromEnum 'a' ) + ( mod ( ( lcOrd c ) + n ) 26 ) )
    | otherwise = c

-- Hilfsfunktion: ist c ein grosser/kleiner Lateinischer Buchstabe?
isULC :: Char -> Bool
isULC c = ( c >= 'A' && c <= 'Z' )
isLLC :: Char -> Bool
isLLC c = ( c >= 'a' && c <= 'z' )

-- Hilfsfunktion zur Bestimmung des Index eines lateinischen Buchstabens
lcOrd :: Char -> Int
lcOrd c
    | ( c >= 'A' && c <= 'Z' ) = fromEnum c - fromEnum 'A'
    | ( c >= 'a' && c <= 'z' ) = fromEnum c - fromEnum 'a'
    | otherwise = error "Expecting Latin character"

{-=============================8<=========================================
========================  Aufgabe 8 =======================================

Gegeben sei eine Funktion f :: a->a und eine Prädikat-Funktion p :: a->Bool:
Definieren Sie eine polymorphe Funktion calculateWhile, die die Funktion f
nur
ab dem ersten Element der Liste, das das Prädikat p erfüllt, bis vor dem
Element
der Liste, das das Prädikat p nicht mehr erfüllt, anwendet.
Anwendungsbeispiel:
calculateWhile (*2) (>0) [-2, -1, 2, 1, 3, 0, -2, 3, 1] ⇒ [-2, -1, 4, 2, 6,
0, -2, 3, 1]

==============================8<=========================================-}
-- (Siehe Funktion contains ganz oben)

calculateWhile :: ( a -> a ) -> ( a -> Bool ) -> [a] -> [a]
```

```
calculateWhile f p xs = calculateWhile2 f p xs False

-- Hilfsfunktion mit Argument, das anzeigt, ob f schon einmal benutzt
--  wurde (um dann nach dem ersten p x == False f nicht mehr anzuwenden)
calculateWhile2 :: ( a -> a ) -> ( a -> Bool ) -> [a] -> Bool -> [a]
calculateWhile2 _ _ [] _ = []
calculateWhile2 f p (x:xs) b =
    if ( p x )
    then ( f x ) : ( calculateWhile2 f p xs True )
    else x : ( if b then xs else ( calculateWhile2 f p xs False ) )

{-==============================8<============================================
========================  Aufgabe 9 ==========================================

Schreiben Sie eine Funktion tokenizer, die aus einem einfachen Text die
Liste
aller Worte des Textes berechnet. Der Text besteht nur aus Buchstaben und
Trennzeichen.
Mit Trennzeichen sind Kommata, Punkte, Fragezeichen und Leerzeichen gemeint.
Die Funktion soll folgende Signatur haben: text2words :: [Char] -> [[Char]]

==============================8<===========================================-}

text2words :: String -> [String]
text2words s = text2words2 s ""

-- Konstante: Liste aus allen Trennzeichen
csDelimiter :: [Char]
csDelimiter = ['\r', '\n', '\t', ' ', ',', '.', ';', ':', '?', '!', '(', ')']

-- Hilfsfunktion mit aktuellem Wort (rueckwaerts) als Argument
text2words2 :: String -> String -> [String]
text2words2 [] "" = []
text2words2 [] w = [reverse w]
text2words2 (c:cs) w
    | ( contains csDelimiter c ) && w /= "" = ( reverse w ) : ( text2words2
cs "" )
    | ( contains csDelimiter c ) = ( text2words2 cs "" )
    | otherwise = text2words2 cs (c:w)
```

Übung 4

```
{-
Uebungszettel 4
Tutorium Katharina Klost 03
17.11.2013
Ludwig Schuster, Benjamin Berendsohn
-}
```

```
{-==============================8<==============================================
========================  Aufgabe 1 ============================================

Die Goldbachsche Vermutung sagt, dass jede gerade Zahl größer als 2 als
Summe
zweier Primzahlen geschrieben werden kann.
a)  Schreiben Sie eine Funktion listOfSums, die unter sinnvoller Verwendung
    von Listengeneratoren, die Liste mit der Summe aller zweier
Zahlenkombinationen
    einer Eingabeliste berechnet. Die Elemente der Liste können mit sich
selber
    kombiniert werden.
    Anwendungsbeispiel:
        listOfSums [1, 2, 0] => [2, 3, 1, 3, 4, 2, 1, 2, 0]
b)  Definieren Sie eine Funktion, die bei Eingabe einer geraden Zahl die
Liste
    aller Goldbachschen Tupel ermittelt. Sie können in Ihrer Definition die
    primzahlen-Funktion aus den Vorlesungsfolien verwenden.
        Anwendungsbeispiel:
            goldbachPairs 32 => [(3, 29), (13, 19)]

==============================8<============================================-}

-- 1 a) Liste der Summen der Zweierpaare.

listOfSums :: [Integer] -> [Integer]
listOfSums [] = []
listOfSums xs = [ x + y | x <- xs, y <- xs ]

-- 1 b) Goldbachsche Vermutung

goldbachPairs :: Integer -> [(Integer, Integer)]
goldbachPairs n
    | ( n <= 2 || mod n 2 == 1 ) = error "Expecting even number greater than
2"
goldbachPairs n = findSumPairs primes n

-- "Ist Vielfaches"-Hilfsfunktion
isMultiple :: Integer -> Integer -> Bool
isMultiple a b = ( mod a b == 0 )

{-==============================8<==============================================
========================  Aufgabe 2 ============================================

Wenn wir eine Menge M mit n verschiedenen Objekten haben, kann die Anzahl
der verschiedenen k-elementigen Teilmengen aus M mit Hilfe des bekannten
Binomialkoeffizienten wie folgt berechnet werden.
Binomialkoeffizient (n, k) =⎛ n ⎞ = n! mit 0 ≤ k ≤ n ⎜⎝k⎟⎠ k!(n−k)!
a)  Schreiben Sie eine Haskell-Funktion mit folgender Signatur
    binom_naiv :: Integer -> Integer -> Integer
```

```
    die mit Hilfe der vorherigen Definition und ohne Rekursion
    (außer innerhalb der Fakultätsfunktion) für beliebige natürliche Zahlen
    n und k den Binomialkoeffizienten berechnet.
    Eine rekursive Definition der gleichen Funktion sieht wie folgt aus:
    ...
    Definieren Sie eine rekursive Haskell-Funktion dafür.
c)  Aus der ersten Definition von a) kann folgende Gleichung abgeleitet
werden:
    ...
    Definieren Sie eine möglichst effiziente Haskell-Funktion, die diese
Definition
    des Binomialkoeffizienten verwendet.
d)  Schreiben Sie eine Test-Funktion, die überprüft, dass alle drei
Funktionen
    das gleiche Ergebnis liefern.

=============================8<=========================================-}

primes :: [Integer]
primes = 2 : primes' 3 [2]

primes' :: Integer -> [Integer] -> [Integer]
primes' n ps = if ( any ( isMultiple n ) ps )
        then ( primes' (n+1) ps )
        else n : ( primes' (n+1) (n:ps) )

-- Allgemeine Funktion zur Bestimmung jedes Paars in
--  einer sortierten Liste, das addiert eine bestimmte
--  Zahl ergibt
findSumPairs :: [Integer] -> Integer -> [(Integer, Integer)]
findSumPairs xs sum = findSumPairs' xs xs sum

-- Die erste Liste bleibt zunaechst gleich und ihr erstes Element
--  wird solange mit den Elementen der zweiten als Paar getestet,
--  bis die Summe des Paares groesser als sum ist (da beide Listen
--  geordnet sind, muss nicht weiter getestet werden). Dann wird mit
--  dem naechsten Element der erste Liste das gleiche gemacht.
--  Die Funktion endet, wenn das gerade betrachtete Element der
--  ersten Liste mit sich selbst addiert schon groesser als sum ist.
findSumPairs' :: [Integer] -> [Integer] -> Integer -> [(Integer, Integer)]
findSumPairs' (x:xs) (y:ys) sum
    | ( x + x > sum ) = []
    | ( x + y > sum ) = ( findSumPairs' xs xs sum ) -- naechstes x
    | ( x + y == sum ) = (x,y) : ( findSumPairs' (x:xs) ys sum ) --
naechstes y
    | ( x + y < sum ) = ( findSumPairs' (x:xs) ys sum ) -- naechstes y
findSumPairs' _ _ _ = [] -- leere Listen abfangen

-- 2 a) Binomialkoeffizient nicht-rekursiv

binom_naiv :: Integer -> Integer -> Integer
```

```
binom_naiv n k
    | ( n < 0 || k < 0 ) = undefined
    | ( k > n ) = 0 {- Aus 2b) -}
binom_naiv n k = div ( faculty n ) ( ( faculty k ) * ( faculty ( n - k ) ) )

faculty :: Integer -> Integer
faculty n
    | ( n < 0 ) = undefined
faculty 0 = 1
faculty n = n * ( faculty ( n - 1 ) )

-- 2 b) Binomialkoeffizient rekursiv

bnRec :: Integer -> Integer -> Integer
bnRec n k
    | ( n < 0 || k < 0 ) = undefined {- Aus 2a) -}
    | ( k > n ) = 0                                           -- (1. Formel)
    | ( k == n ) = 1                                       -- (2.)
    | ( k == 0 ) = 1                                       -- (2.)
    | ( k == 1 ) = n                                       -- (3.)
    | ( k == n-1 ) = n                                       -- (3.)
bnRec n k =    ( bnRec (n-1) (k-1) ) + ( bnRec (n-1) k )    -- (4.)

-- 2 c) Nochmal Binomialkoeffizient

bn2 :: Integer -> Integer -> Integer
bn2 n k
    | ( n < 0 || k < 0 ) = undefined {- Aus 2a) -}

bn2 _ 0 = 1
bn2 0 _ = 0
bn2 n k = bn2' 1 1 n k

bn2' :: Integer -> Integer -> Integer -> Integer -> Integer
-- r ist das "zwischengespeicherte" Resultat
-- i ist die Nummer des "Schleifendurchlaufs", 1 <= i <= k
-- n k sind Konstanten, die Eingabewerte fuer bn2
bn2' r i n k = if ( i == k ) then a else bn2' a (i+1) n k
    where
        a = div ( r * ( n - i + 1 ) ) i

-- Bei der Definition von a darf "div" statt "/" benutzt werden, da das
--  Zwischenresultat a immer gleich ( n! / (n-i)! ) / i! ist.
--  Bsp. i=1: ( n / 1 ) = ( n! / (n-1)! ) / 1!
--  Bsp. i=2: ( n / 1 ) * ( (n-1) / 2 ) = n * (n-1) / 2 = ( n! / (n-2)! ) /
2!
--  ...
--  ( n! / (n-i)! ) / i! = "n ueber i", was eine ganze Zahl ist. a ist also
--  eine ganze Zahl, und damit auch, im naechsten Schleifendurchlauf, r.
--  Weiter sind n und i auch ganze Zahlen, daher auch ( r * ( n - i + 1 ) ).
--  Somit sind beide Argumente sowie das Ergebnis der Division ganze Zahlen
```

```
--  und die div-Funktion verursacht keine Rundung.

-- 2 d) Test-Funktion
-- Testet alle Werte bis max
testBnAll :: Integer -> Bool
testBnAll max = all ( testBnAll' max ) [0..max]

-- Funktion mit vorgegebenem n
testBnAll' :: Integer -> Integer -> Bool
testBnAll' max n = all ( testBn n ) [0..max]

-- Testet einen einzelnen Wert
testBn :: Integer -> Integer -> Bool
testBn n k
    | ( n < 0 || k < 0 ) = undefined
testBn n k = ( binom_naiv n k ) == ( bnRec n k ) && ( binom_naiv n k ) == (
bn2 n k )

{-============================8<==========================================
========================  Aufgabe 3 ======================================

Definieren Sie eine polymorphe Funktion positions, die die Positionen aller
vorkommenden Elemente innerhalb einer Liste wiederum in einer Liste
zurückgibt.
    Anwendungsbeispiel:
        positions 'a' "Maria Antonieta" => [1, 4, 14]
a)  Definieren Sie zuerst die Funktion unter Verwendung von expliziter
    Rekursion und Akkumulator-Technik.
b)  Definieren Sie die Funktion unter sinnvoller Verwendung von
Listengeneratoren
    und mindestens einer Funktion höherer Ordnung.

==============================8<========================================-}



-- 3 a) positions-Funktion mit Akkumulatortechnik
positions :: (Ord a) => a -> [a] -> [Integer]
positions e es = reverse ( positions' e es 0 [] )

positions' :: (Ord a) => a -> [a] -> Integer -> [Integer] -> [Integer]
positions' e [] i ps = ps
positions' e (e2:es) i ps = positions' e es (i+1) ( if e2 == e then (i:ps)
else ps )

-- 3 b) positions-Funktion mit Listengenerator und Funktionen hoeherer
Ordnung

positions3 :: (Ord a) => a -> [a] -> [Integer]
positions3 e xs = reverse [p | ( e2, p ) <- ( index xs ), e2 == e ]

-- Hilfsfunktion: Gibt eine Liste mit einem Paar aus jedem Element und
```

```
--   seiner Position zurueck.
index :: [a] -> [(a, Integer)]
index es = foldl index' [] es

index' :: [(a, Integer)] -> a -> [(a, Integer)]
index' [] e = [(e, 0)]
index' (x:xs) e = ( e, (snd x ) + 1 ) : x : xs

{-============================8<=============================================
========================  Aufgabe 4 =======================================

In dieser Aufgabe sollen drei Funktionen mit folgende Namen definiert werden,
die die untenstehenden Zeichenbilder ausgeben.

FARM                                SQUARES
..............................      .............................0
..............................       ...........................00
..............................        .........................000
..............................         .......................0000
..............................          .....................00000
..............................           ...................000000
..............................            .................0000000
.......000000000000000........             ...............00000000
....... 00000000000000........              .............000000000
.......  0000000000000........               ...........0000000000
.......   000000000000........                .........00000000000
.......    00000000000........                 .......000000000000
.......     0000000000........                  .....0000000000000
.......      000000000........                   ...00000000000000
.......       00000000........                    .000000000000000
#######        0000000########      ************** 000000000000000
#######         000000########      **************  00000000000000
#######          00000########      **************   0000000000000
#######           0000########      **************    000000000000
#######            000########      **************     00000000000
#######             00########      *****     ****      0000000000
#######              0########      *****     ****       000000000
##############################      *****     ****        00000000
##############################      *****     ****         0000000
##############################      *****     ****          000000
##############################      **************           00000
##############################      **************            0000
##############################      **************             000
##############################      **************              00
##############################      **************               0

SQUARES                                     CIRCLE
                   |                                 ...........
                  |||                              ...............
                 |||||                           ...................
                |||||||                         .....................
```

```
CHESSBOARD
     00000     00000     00000     00000
     00000     00000     00000     00000
     00000     00000     00000     00000
     00000     00000     00000     00000
     00000     00000     00000     00000
00000     00000     00000     00000
00000     00000     00000     00000
00000     00000     00000     00000
00000     00000     00000     00000
00000     00000     00000     00000
     00000     00000     00000     00000
     00000     00000     00000     00000
     00000     00000     00000     00000
```

```
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
     OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO
OOOOO     OOOOO     OOOOO     OOOOO

Eine paintPicture Funktion, die als Argument eine dieser Funktionsnamen
bekommt,
wird vorgegeben (siehe im KVV). Diese Funktion darf nicht verändert werden.
Die drei zu implementierenden Funktionen bekommen jeweils als Argumente ein
(x, y, size) Tupel, das aus den x, y- Koordinaten und size, der Seitenlänge
des Bildes, besteht, und entscheidet, welches Zeichen an einer bestimmten
Position
zurückgegeben wird. Innerhalb der zu implementierenden Funktionen darf keine
Rekursion
verwendet werden. Die printPicture Funktion, so wie zwei Beispielfunktionen
sind
auf der Veranstaltungsseite unter Ressourcen herunter zu laden.

==============================8<========================================-}



-- Testfunktionen
testFarm :: IO ()
testFarm = putStrLn ( paintPicture farm 30 )

testSquares :: IO ()
testSquares = putStrLn ( paintPicture squares 30 )
```

```
testDiamond :: IO ()
testDiamond = putStrLn ( paintPicture diamond 40 )

testCircle :: IO ()
testCircle = putStrLn ( paintPicture circle 30 )

testChessboard :: IO ()
testChessboard = putStrLn ( paintPicture chessboard 40 )

-- Funktion aus der Vorlesung

paintPicture :: ((Int, Int, Int) -> Char) -> Int -> [Char]
paintPicture f size = paint size (map f [(x,y,size) | x <- [1..size], y <- [
1..size]])
    where
        paint 0  []     = []
        paint 0 (c:cs)  = '\n' : (paint size (c:cs))
        paint n (c:cs)  = c: (paint (n-1) cs)

-- 4a) farm
farm :: (Int, Int, Int) -> Char
farm ( x, y, size ) =
    if x > ( div size 4 ) && x <= ( div (3*size) 4 ) && y > ( div size 4 )
&& y <= ( div (3*size) 4 )
    then ( if y >= x then '0' else ' ' )
    else ( if x > ( div size 2 ) then '#' else '.' )

-- 4b) squares
squares :: (Int, Int, Int) -> Char
squares ( x, y, size ) =
    if (y-x) >= 0
    then
        if (x+y) <= size then '.' else 'O'
    else
        if x > div size 2 && y < div size 2
            && ( ( x <= div (4*size) 6 || x > div (5*size) 6 )
                || ( y <= div (1*size) 6 || y > div (2*size) 6 ) )
        then '*'
        else ' '

-- 4c) diamond
diamond :: (Int, Int, Int) -> Char
diamond ( x, y, size ) =
    if (x+y) > ( div size 2 ) && (x+y) < ( div (3*size) 2 )
        && (y-x) < ( div size 2 ) && (x-y) < ( div size 2 )
    then '|' else ' '

-- 4d) circle
circle :: (Int, Int, Int) -> Char
circle ( x, y, size ) =
    if ( ( x - div size 2 )^2 + ( y - div size 2 )^2 <= ( div (size^2) 4 ) )
```

```
then '.' else ' '

-- 4e) chessboard
chessboard :: (Int, Int, Int) -> Char
chessboard ( x, y, size ) = if mod ( gridX + gridY ) 2 == 1 then 'O' else '
'
    where
        gridX = div ( (x-1) * 8 ) size
        gridY = div ( (y-1) * 8 ) size
```

Übung 5

```
{-
Uebungszettel 4
Tutorium Katharina Klost 03
24.11.2013
Ludwig Schuster, Benjamin Berendsohn
-}

-- 0) Testfunktionen

-- Algemeine Testfunktion
unaryTest :: (Show a, Show b, Eq b) => [(a, b)] -> ( a -> b ) -> String ->
Bool
unaryTest [] _ _ = True
unaryTest ((x,y):ts) f s =
    if( y /= f x )
    then error ( s ++ ": test failed on input " ++ show x ++ ": expected ("
++
        show y ++ "), got (" ++ show (f x) ++ ")." )
    else
        unaryTest ts f s

binaryTest :: (Show a, Show b, Show c, Eq c) => [((a,b),c)] -> ( a -> b -> c )
-> String -> Bool
binaryTest ts f s = unaryTest ts ( uncurry f ) s

-- Testfunktionen
allTest :: Bool
allTest = longestRepSeqTest && flatten1Test && flatten2Test
        && bin2decTest && majorityTest

longestRepSeqTest :: Bool
longestRepSeqTest = unaryTest longestRepSeqTestData longestRepSeq
"longestRepSeq"
flatten1Test :: Bool
flatten1Test = unaryTest flattenTestData flatten1 "flatten1"
flatten2Test :: Bool
```

```
flatten2Test = unaryTest flattenTestData flatten2 "flatten2"
bin2decTest :: Bool
bin2decTest = unaryTest bin2decTestData bin2dec "bin2dec"
majorityTest :: Bool
majorityTest = unaryTest majorityTestData majority "majority"

-- Testdaten
longestRepSeqTestData :: [( [Integer], [Integer] )]
longestRepSeqTestData = [
    ( [1,1,0,1,1,0,0,1,0,1,1] , [1,0,1,1] ),
    ( [1,1,0,1,1,0,0,1,0,1] , [1,1,0] ),
    ( [0] , [0] )
    ]

flattenTestData :: [([String], String )]
flattenTestData = [
    ( ["abc", "bcd", "abc" ], "abcbcdabc" ),
    ( ["abc", "de", "f" ], "abcdef" ),
    ( ["", "a", "", "b" ], "ab" ),
    ( ["a"], "a" ),
    ( [], "" )
    ]

bin2decTestData :: [([Int],Int)]
bin2decTestData = [
    ( [1,0,0,1], 9 ),
    ( [1,1,0,1], 13 ),
    ( [0], 0 ),
    ( [1], 1 ),
    ( [], 0 )
    ]

majorityTestData :: [([Int], Maybe Int)]
majorityTestData = [
    ( [1], Just 1 ),
    ( [0,1,1], Just 1 ),
    ( [0,1,0,1,0,1,1], Just 1 ),
    ( [1,1,0,0,0,1,1], Just 1 ),
    ( [0,1,1,1,0], Just 1 ),
    ( [1,1,1,1,1,0,0,0,0], Just 1 ),
    ( [1,1,1,1,0,0,0,0], Nothing ),
    ( [], Nothing )
    ]
{-===========================8<=========================================
========================  Aufgabe 1 =====================================

Definieren Sie die Funktion longestRepSeq, die die längste sich
wiederholende
Objektsequenz aus einer Objekt-Liste findet (siehe Vorlesungsfolien).
Die Funktion soll folgende Signatur haben.
    longestRepSeq :: (Ord a) => [a] -> [a]
```

```
    Anwendungsbeispiele:
        longestRepSeq "absdadsdahko" => "sda"
         longestRepSeq [1,1,0,1,1,0,0,1,0,1,1] => [1,0,1,1]
a)  Programmieren Sie zuerst folgende drei Hilfsfunktionen:
    i)  Eine Funktion listOfSuffixes, die alle verschiedenen Suffixe aus
einer
        Zeichenkette in einer Liste als Ergebnisse zurückgibt.
    Anwendungsbeispiel:
        listOfSuffixes "xyzab" => ["xyzab","yzab","zab","ab","b"]
    ii) Definieren Sie eine Funktion prefix, die zwei Zeichenketten bekommt
        und das längste gemeinsame Prefix, falls es eines gibt, berechnet.
    Anwendungsbeispiel:
        prefix "abcde" "abacde" => "ab"
    iii)Definieren Sie eine Funktion longestPrefix, die eine Liste von
        Zeichenketten durchgeht und das längste Prefix aus zwei benachbarten
        Zeichenketten der Liste in einem Tupel als Ergebnis zurückliefert.
        Das erste Element des Tupels ist die Länge des Prefixes.
    Anwendungsbeispiel:
        longestPrefix ["a","abca","bca","bcadabca","ca","cdabca"] =>
(3,"bca")
b)  Programmieren Sie dann unter Verwendung Ihrer Funktionen in i), ii), iii)
    und ein geeignete Sortieralgorithmus die longestRepSeq Funktion.
    Verwenden Sie Funktionskomposition in Ihrer Definition.
c)  Analysieren Sie die Komplexität Ihrer Funktionen.

==============================8<==========================================-}

-- 1a i) listOfSuffixes

listOfSuffixes :: [a] -> [[a]]
listOfSuffixes [] = []
listOfSuffixes (x:xs) = (x:xs) : listOfSuffixes xs

-- 1a ii) prefix

prefix :: (Eq a) => [a] -> [a] -> [a]
prefix [] _ = []
prefix _ [] = []
prefix (x:xs) (y:ys) =
    if ( x == y )
    then x : ( prefix xs ys )
    else []

-- 1a iii) longestPrefix

longestPrefix:: (Eq a) => [[a]] -> ( Int, [a] )
longestPrefix [] = ( 0, [] )
longestPrefix [xs] = ( length xs, xs )
longestPrefix (xs1:xs2:xss) = fstsnd ( foldl longestPrefix2 (length p, p,
xs2) xss )
    where
```

```
        p = prefix xs1 xs2

longestPrefix2 :: (Eq a) => ( Int, [a], [a] ) -> [a] -> ( Int, [a], [a] )
-- l: Laenge des bisher laengsten Prefixes
-- p1: Bisher laengstes Prefix
-- xs1: Letzte iterierte Unterliste
-- xs2: Gerade iterierte Unterliste
longestPrefix2 (l, p1, xs1) xs2 = if ( l > length p2 ) then (l, p1, xs2)
else (length p2, p2, xs2)
    where
        p2 = prefix xs1 xs2

-- Hilfsfunktion: Erste beide Elemente eines Tupels mit drei Elementen
fstsnd :: ( a, b, c ) -> ( a, b )
fstsnd ( x, y, z ) = ( x, y )

-- 1b) longestRepSeq

longestRepSeq :: (Ord a) => [a] -> [a]
longestRepSeq = snd . longestPrefix . startMergeSort . listOfSuffixes

-- Mergesort-Implementierung aus der Vorlesung
--  Komplexitaet: O( n*log(n) )
split :: [a] -> [[a]]
split [] = []
split [x] = [[x]]
split (x:xs) = [x]: (split xs)

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
    | x <= y = x: (merge xs (y:ys))
    | otherwise= y: (merge (x:xs) ys)

mergeLists :: (Ord a) => [[a]] -> [[a]]
mergeLists [] = []
mergeLists [x] = [x]
mergeLists (x:y:xs) = (merge x y): mergeLists xs

mergeSort :: (Ord a) => [[a]] -> [[a]]
mergeSort [x] = [x]
mergeSort (x:y:xs) = mergeSort (mergeLists (x:y:xs))

startMergeSort :: (Ord a) => [a] -> [a]
startMergeSort xs = sortedList
    where
        [sortedList] = mergeSort (split xs)

{-============================8<==========================================
========================  Aufgabe 2 ========================================
```

```
Definieren Sie eine Haskell-Funktion flatten :: [[ a]] -> [a], welche eine
Liste von Listen zu einer Liste kombiniert.
    Anwendungsbeispiel:
        flatten ["abc", "bcd", "abc"] => "abcbcdabc"
a)  Schreiben Sie zuerst Ihre Definition mit Hilfe der foldr-Funktion.
b)  Schreiben Sie eine zweite Definition mit der foldl-Funktion.
c)  Welche der beiden Lösungen ist besser? Warum?

=============================8<==========================================-}

-- 2 a) flatten mit foldr
flatten1 :: [[a]] -> [a]
flatten1 xss = foldr (++) [] xss

{- Entspricht etwa:
flatten1 [] = []
flatten1 (x:xs) = x ++ (flatten1 xs)
-}

-- 2 b) flatten mit foldl

flatten2 :: [[a]] -> [a]
flatten2 xss = foldl (++) [] xss

{- Entspricht etwa:
flatten2 = flatten22 []
flatten22 z [] = z
flatten22 z (x:xs) = flatten22 (z ++ x) xs
-}

{-============================8<===========================================
========================  Aufgabe 3 ======================================

Definieren Sie unter Verwendung der foldl-Funktion noch einmal die bin2dec
Funktion aus der Vorlesung, die als Eingabe eine Liste von Bits bekommt und
daraus die entsprechende Dezimal-Zahl berechnet.

=============================8<==========================================-}

bin2dec :: [Int] -> Int
bin2dec xs = foldl nextBin2dec 0 xs

nextBin2dec :: Int -> Int -> Int
nextBin2dec x c
    | ( c /= 0 && c /= 1 ) = error "Not a binary digit"
nextBin2dec x c = 2 * x + c

{-============================8<===========================================
========================  Aufgabe 4 ======================================
```

```
Ein Element einer Liste von n Objekten stellt die absolute Mehrheit der
Liste dar,
wenn das Element mindestens [...] mal in der Liste vorkommt. Definieren Sie
eine majority-Funktion, die mit linearem Aufwand das Majority-Element der
Liste findet,
wenn eines existiert oder sonst Nothing zurückgibt. Die Signatur der
Funktion soll wie
folgt aussehen:
    majority :: (Eq a) => [a] -> Maybe a

==============================8<=========================================-}

majority :: (Eq a) => [a] -> Maybe a
majority [] = Nothing
majority (x:xs) =
    if ( c >= ( ( div ( length (x:xs) ) 2 ) + 1 ) )
    then Just m
    else Nothing
    where
        m = fst ( foldl nextMajority (x,0) xs )
        c = foldl ( nextCount m ) 0 (x:xs)
        -- c = length [y|y <- (x:xs), y == m] ist schoener,
        -- aber ineffizienter

-- Hilfsfunktion zum Zaehlen der Vorkommnisse der Elemente
nextCount :: (Eq a) => a -> Int -> a -> Int
nextCount x1 i x2 =
    if ( x1 == x2 )
    then i + 1
    else i

-- Hilfsfunktion mit der Methode aus der Vorlesung
nextMajority :: (Eq a) => ( a, Int ) -> a -> ( a, Int )
nextMajority ( x1, i ) x2 =
    if ( x1 == x2 )
    then ( x1, i+1 )
    else if( i > 0 ) then ( x1, i-1 ) else ( x2, 0 )

-- TODO: Beweis linearer Aufwand
```

Übung 6

```
{-
Uebungszettel 5
Tutorium Katharina Klost 03, Fr 14-16 Uhr
1.12.2013
Ludwig Schuster, Benjamin Berendsohn
-}
```

```
---------------------------------------------
------ Imports und Typdefinitionen
---------------------------------------------
import QRationals

data SBTree = L | N SBTree SBTree
    deriving Show

data BSearchTree a = Nil| Node a (BSearchTree a) (BSearchTree a)
    deriving( Show, Eq )

-- Hinzugefuegte Funktionen

instance Eq Nat where
    (==) Zero Zero = True
    (==) (S a) (S b) = ( a == b )
    (==) _ _ = False

instance Eq ZInt where
    (==) (Z a b) (Z c d) = (add a d) == (add b c)

instance Eq SBTree where
    (==) L L = True
    (==) (N l1 r1) (N l2 r2) = ( l1 == l2 ) && ( r1 == r2 )
    (==) _ _ = False

{-=============================8<=============================================
========================  Aufgabe 1 ==========================================

Definieren Sie folgende Funktionen für den algebraischen Datentyp Nat aus
der Vorlesung.
        ngerade :: Nat -> Bool nmax :: Nat -> Nat -> Nat
b)  Programmieren Sie folgende Hilfsfunktionen, die das Testen der
Funktionen
    vereinfachen sollen.
        nat2Int :: Nat -> Integer int2Nat :: Integer -> Nat
    Anwendungsbeispiel:
        nat2Int (S (S (S Zero))) => 3 int2Nat 5 => (S (S (S (S (S Zero)))))
c)  Programmieren Sie folgende Funktionen für den algebraischen Datentyp
ZInt
    aus der Vorlesung.
        zpow :: ZInt -> Nat -> ZInt                      -- Potenz-Funktion
       zabs :: ZInt -> ZInt                             -- Absoluter Wert
       zggt :: ZInt -> ZInt --> ZInt                    -- Größter
gemeinsamer Teiler
d)  Definieren Sie folgende Hilfsfunktionen.
        zint2Int :: ZInt -> Integer -- transformiert von ZInt nach Integer
        int2Zint :: Integer -> ZInt -- transformiert von Integer nach ZInt

==============================8<===========================================-}
```

```
------ 1a) Gerade und Maximum

-- Eine Zahl ist genau dann gerade, wenn ihr Vorgaenger
--  nicht gerade ist
ngerade :: Nat -> Bool
ngerade Zero = True
ngerade (S n) = not ( ngerade n )

nmax :: Nat -> Nat -> Nat
nmax n m = nmax2 n m n m

-- Hilfsfunktion: Speichert die Argumente, um eins von
-- beiden am Ende zurueckgeben zu koennen
nmax2 :: Nat -> Nat -> Nat -> Nat -> Nat
nmax2 n m Zero _ = m
nmax2 n m _ Zero = n
nmax2 n m (S x) (S y) = nmax2 n m x y

------ 1b) Konvertierung Nat <-> Integer

-- Eine Zahl ist um eins groesser als ihr Vorgaenger

nat2Int :: Nat -> Integer
nat2Int Zero = 0
nat2Int (S n) = 1 + nat2Int n

int2Nat :: Integer -> Nat
int2Nat 0 = Zero
int2Nat n =
    if ( n >= 1 )
    then S ( int2Nat (n-1) )
    else error ( "int2Nat: invalid input: " ++ show n )

------ 1c) Potenz, absoluter Wert, ggt von ZInt
zpow :: ZInt -> Nat -> ZInt
zpow a Zero = zone
zpow a ( S b ) = zmult a ( zpow a b )

zabs :: ZInt -> ZInt
-- Wandelt natuerliche in ganze Zahl um
zabs i = Z Zero ( znabs i )

-- Euklidischer Algorithmus
zggt :: ZInt -> ZInt -> ZInt
-- Der ggT kann mit Betraegen berechnet werden, da eine Zahl
--  und ihre Negation die gleichen Teiler hat.
zggt a b = Z Zero ( zggt2 ( znabs a ) ( znabs b ) )

zggt2 :: Nat -> Nat -> Nat
zggt2 a b = if ( r == Zero ) then b
```

```
    else zggt2 b r
    where
        (_, r) = nDivRem a b

-- Hilfsfunktion: Teilung mit Rest
nDivRem :: Nat -> Nat -> (Nat, Nat)
nDivRem _ Zero = error "Division by Zero"
nDivRem a b = nDivRem2 a b (Zero, Zero)

-- Hilfsgunktion : Der Divisor b bleibt immer gleich. Es
--  wird vom Dividenten a in jeden Durchlauf eins abgezogen
--  und auf den Rest r aufaddiert. Wenn der Rest den Divisor
--  erreicht hat, wird der Quotient q um eins erhoeht und
--  der Rest auf 0 zurueckgesetzt. Wenn der Divident 0
--  erreicht, hat ist er vollstaendig auf den Quotienten und
--  Rest aufgeteilt.
nDivRem2 :: Nat -> Nat -> (Nat, Nat) ->(Nat, Nat)
nDivRem2 a b (q,r)
    | ( r == b ) = nDivRem2 a b ( S q, Zero )
nDivRem2 Zero b (q,r) = (q,r)
nDivRem2 (S a) b (q,r) = nDivRem2 a b ( q, S r )

-- Hilfsfunktion: Betrag einer ganzen Zahl als
--  natuerliche Zahl
znabs :: ZInt -> Nat
-- Gibt etweder nur den negativen oder nur den positiven
--  Teil zurück
znabs i =
    if ( p /= Zero ) then p
    else n
    where
        Z n p = zsimplify i

------ 1c) Konvertierung ZInt <-> Integer
zint2Int :: ZInt -> Integer
zint2Int (Z n Zero) = - ( nat2Int n )
zint2Int (Z Zero p) = nat2Int p
-- entspricht simplify
zint2Int ( Z (S n) (S p) ) = zint2Int (Z n p)

int2Zint :: Integer -> ZInt
int2Zint i =
    if ( i > 0 )
    then ( Z Zero (int2Nat i) )
    else ( Z ( int2Nat (-i) ) Zero )

{-===========================8<=========================================
=======================  Aufgabe 2 ======================================
```

```
==============================8<=========================================-}

-- Um den Baum moeglichst balanciert zu halten, wird das am wenigsten
--  Tiefe Blatt durch einen Knoten mit zwei Blaettern ersetzt.
insertLeaf :: SBTree -> SBTree
insertLeaf L = N L L
insertLeaf (N t1 t2) =
    -- Links wird bevorzugt eingefuegt
    if ( minDepth t1 <= minDepth t2 )
    then N (insertLeaf t1) t2
    else N t1 (insertLeaf t2)

insertLeafs :: SBTree -> Integer -> SBTree
insertLeafs _ i
    | ( i < 0 ) = error "Can't insert negative amount of leaves"
insertLeafs t 0 = t
insertLeafs t i = insertLeaf (insertLeafs t (i-1))

-- Um den Baum moeglichst balanciert zu halten, wird der tiefste Knoten
--  durch ein Blatt ersetzt
deleteLeaf :: SBTree -> SBTree
deleteLeaf L = error "Can't delete a single leaf"
deleteLeaf (N L L) = L
deleteLeaf (N t1 t2) =
    -- Rechts wird bevorzugt geloescht
    if ( height t1 > height t2 )
    then N (deleteLeaf t1) t2
    else N t1 (deleteLeaf t2)

deleteLeafs :: SBTree -> Integer -> SBTree
deleteLeafs _ i
    | ( i < 0 ) = error "Can't delete negative amount of leaves"
deleteLeafs t 0 = t
deleteLeafs t i = deleteLeaf (deleteLeafs t (i-1))

-- Ein Binaerer Baum ist vollstaendig, wenn alle Bl"atter die
--  gleiche Tiefe haben.
full :: SBTree -> Bool
full L = True
full (N t1 t2) = full t1 && full t2 && ( height t1 == height t2 )

-- Hilfsfunktion: Minimale Tiefe unter allen Blaettern
minDepth :: SBTree -> Integer
minDepth L = 0
minDepth (N t1 t2) = 1 + ( min (minDepth t1) (minDepth t2) )

-- Hilfsfunktion: Hoehe eines Baumes, entspricht maximaler Tiefe
--  unter allen Blaettern
height :: SBTree -> Integer
height L = 0
height (N t1 t2) = 1 + max ( height t1 ) ( height t2 )
```

```
{-============================8<========================================
========================  Aufgabe 3 ====================================



============================8<=======================================-}

---- 3 a)

twoChildren :: (Ord a) => BSearchTree a -> Bool
twoChildren Nil = True -- Kein Knoten vorhanden.
twoChildren (Node _ Nil Nil) = True -- Blatt
twoChildren (Node _ _ Nil) = False -- Nur ein Kind
twoChildren (Node _ Nil _) = False -- Nur ein Kind
twoChildren (Node _ t1 t2) = twoChildren t1 && twoChildren t2

-- Wendet Funktion auf alle Werte im Baum an. Baum muss danach nicht
--  sortiert sein.
mapTree :: (Ord a, Ord b) => (a -> b) -> BSearchTree a -> BSearchTree b
mapTree f Nil = Nil
mapTree f (Node x l r) = ( Node (f x) (mapTree f l) (mapTree f r) )

-- Wendet Funktion auf Wert des Wurzelknotens und Wert der beiden
-- Kindbaeume an. Leere Baeume haben den voregegebenen Wert.
foldTree :: (Ord a) => b -> (a -> b -> b -> b) -> BSearchTree a -> b
foldTree z f Nil = z
foldTree z f (Node x l r) = f x (foldTree z f r) (foldTree z f l)

--- 3 b)

-- Anwendung auf BSearchTree statt SBTree, wie in der Vorlesung

-- Nil-"Knoten" haben Tiefe (-1), damit ein Blatt
-- ( max (-1), (-1) ) + 1 = 0 Tiefe hat.
depth :: (Ord a) => BSearchTree a -> Integer
depth t = foldTree (-1) depth2 t

depth2 :: a -> Integer -> Integer -> Integer
depth2 _ d1 d2 = ( max d1 d2 ) + 1

-- Entspricht "nodes"-Funktion aus der Vorlesung: Blaetter werden
--  nicht mitgezaehlt.
size :: (Ord a) => BSearchTree a -> Integer
size t = foldTree 0 size2 t

size2 :: a -> Integer -> Integer -> Integer
size2 _ s1 s2 = s1 + s2 + 1



---------------------------------------------
```

```haskell
------ Tests
---------------------------------------------

---- Testdaten

-- 1)

ngeradeTestData :: [(Nat, Bool)]
ngeradeTestData = [
    ( Zero, True ),
    ( one, False ),
    ( two, True ),
    ( nine, False ),
    ( ten, True )
    ]

nmaxTestData :: [((Nat, Nat), Nat)]
nmaxTestData = [
    ( ( Zero, Zero ), Zero ),
    ( ( one, Zero ), one ),
    ( ( two, nine ), nine ),
    ( ( ten, Zero ), ten )
    ]

nat2IntTestData :: [(Nat, Integer)]
nat2IntTestData = [
    ( Zero, 0 ),
    ( one, 1 ),
    ( two, 2 ),
    ( nine, 9 ),
    ( ten, 10 )
    ]

int2NatTestData :: [(Integer, Nat)]
int2NatTestData = [
    ( 0, Zero ),
    ( 1, one ),
    ( 5, five ),
    ( 8, eight ),
    ( 9, nine )
    ]

zpowTestData :: [((ZInt, Nat), ZInt)]
zpowTestData = [
    ( ( zone, ten ), zone ),
    ( ( ztwo, three ), zeight ),
    ( ( zthree, two ), znine ),
    ( ( mztwo, three ), mzeight ),
    ( ( mztwo, two ), zfour )
    ]
```

```
zabsTestData :: [(ZInt, ZInt)]
zabsTestData = [
    ( zone, zone ),
    ( mzone, zone ),
    ( ztwo, ztwo ),
    ( mztwo, ztwo )
    ]

zggtTestData :: [((ZInt, ZInt), ZInt)]
zggtTestData = [
    ( ( zone, zten ), zone ),
    ( ( ztwo, zthree ), zone ),
    ( ( zthree, zsix ), zthree ),
    ( ( mzfour, zsix ), ztwo ),
    ( ( zeight, mzfour ), zfour )
    ]

zint2IntTestData :: [(ZInt, Integer)]
zint2IntTestData = [
    ( Z Zero Zero, 0 ),
    ( zone, 1 ),
    ( znine, 9 ),
    ( mztwo, (-2) ),
    ( mzfive, (-5) )
    ]

int2ZintTestData :: [(Integer, ZInt)]
int2ZintTestData = [
    ( 0, Z Zero Zero ),
    ( 1, zone ),
    ( 9, znine ),
    ( (-2), mztwo ),
    ( (-5), mzfive )
    ]

-- 2)

minDepthTestData :: [(SBTree, Integer)]
minDepthTestData = [
    ( L, 0 ),
    ( N L L, 1 ),
    ( N ( N L L ) L, 1 ),
    ( N ( N L L ) ( N L L ), 2 )
    ]

heightTestData :: [(SBTree, Integer)]
heightTestData = [
    ( L, 0 ),
    ( N L L, 1 ),
    ( N ( N L L ) L, 2 ),
    ( N ( N L L ) ( N L L ), 2 )
```

```
    ]

insertLeafTestData :: [(SBTree, SBTree)]
insertLeafTestData = [
    ( L, N L L ),
    ( N L L, N ( N L L ) L ),
    ( N ( N L L ) L, N ( N L L ) ( N L L ) ),
    ( N ( N L L ) ( N L L ), N ( N ( N L L ) L ) ( N L L ) )
    ]

insertLeafsTestData :: [((SBTree, Integer), SBTree)]
insertLeafsTestData = [
    ( (L, 0), L ),
    ( (L, 1), N L L ),
    ( (L, 2), N ( N L L ) L ),
    ( (L, 3), N ( N L L ) ( N L L ) ),
    ( (N L L, 1), N ( N L L ) L ),
    ( (N L L, 2), N ( N L L ) ( N L L ) ),
    ( (N L L, 3), N ( N ( N L L ) L ) ( N L L ) ),
    ( (N L L, 2), insertLeafs (L) 3 )
    ]

deleteLeafTestData :: [(SBTree, SBTree)]
deleteLeafTestData = [
    ( N L L, L ),
    ( N ( N L L ) L, N L L ),
    ( N ( N L L ) ( N L L ), N ( N L L ) L ),
    ( N ( N ( N L L ) L ) ( N L L ), N ( N L L ) ( N L L ) )
    ]

deleteLeafsTestData :: [((SBTree, Integer), SBTree)]
deleteLeafsTestData = [
    ( (L, 0), L ),
    ( (N L L, 1), L ),
    ( (N ( N L L ) L, 2), L ),
    ( (N ( N L L ) ( N L L ), 3), L ),
    ( (N ( N L L ) L, 1), N L L ),
    ( (N ( N L L ) ( N L L ), 2), N L L ),
    ( (N ( N ( N L L ) L ) ( N L L ), 3), N L L ),
    ( (N ( N L L ) ( N L L ), 2), deleteLeafs (N ( N L L ) L) 1 )
    ]

fullTestData :: [(SBTree, Bool)]
fullTestData = [
    ( L, True ),
    ( N L L, True ),
    ( N ( N L L ) L, False ),
    ( N ( N L L ) ( N L L ), True )
    ]

-- 3 a)
```

```
twoChildrenTestData :: [(BSearchTree Int, Bool)]
twoChildrenTestData = [
    ( Nil, True ),
    ( Node 1 (leaf 1) Nil, False ),
    ( Node 1 Nil (leaf 1), False ),
    ( Node 1 (leaf 1) (leaf 1), True )
    ]

mapTreeTestData :: [( ( (Int -> Int), BSearchTree Int ), BSearchTree Int )]
mapTreeTestData = [
    ( ( (+1), (Node 0 (leaf 1) Nil) ),
        (Node 1 (leaf 2) Nil) ),
    ( ( (^2), ( Node 1 (leaf 2) (leaf 3) ) ),
        ( Node 1 (leaf 4) (leaf 9) ) )
    ]

foldTreeTestData :: [( ( Int, (Int -> Int -> Int -> Int), BSearchTree Int ),
Int )]
foldTreeTestData = [
    ( ( 0, testFuncSum, (Node 1 (leaf 2) Nil) ), 3 ),
    ( ( 1, testFuncSum, ( Node 1 (leaf 3) (leaf 5) ) ), 13 ),
    ( ( 1, testFuncProd, ( Node 2 (leaf 2) (leaf 4) ) ), 16 ),
    ( ( 2, testFuncProd, ( Node 3 (leaf 3) Nil ) ), 72 )
    ]

-- 3 b)

depthTestData :: [( BSearchTree Int, Integer )]
depthTestData = [
    ( (leaf 0), 0 ),
    ( (Node 0 (leaf 0) Nil), 1 ),
    ( ( Node 1 ( Node 2 (leaf 7) (leaf 1) ) (leaf 5) ), 2 )
    ]

sizeTestData :: [( BSearchTree Int, Integer )]
sizeTestData = [
    ( (leaf 0), 1 ),
    ( (Node 0 (leaf 0) Nil), 2 ),
    ( ( Node 1 ( Node 2 (leaf 7) (leaf 1) ) (leaf 5) ), 5 )
    ]

-- Unter-Testfunktionen
testFuncSum :: Int -> Int -> Int -> Int
testFuncSum a b c = a + b + c

testFuncProd :: Int -> Int -> Int -> Int
testFuncProd a b c = a * b * c

-- Helferfunktion zur Datenkonstruktion
leaf :: (Ord a) => a -> BSearchTree a
```

```
leaf x = Node x Nil Nil

---- Allgemeine Testfunktionen
unaryTest :: (Show a, Show b, Eq b) =>
    [(a, b)] -> ( a -> b ) -> String -> Bool
unaryTest [] _ _ = True
unaryTest ((x,y):ts) f s =
    if( y /= f x )
    then error ( s ++ ": test failed on input " ++ show x
        ++ ": expected (" ++ show y ++ "), got ("
        ++ show (f x) ++ ")." )
    else
        unaryTest ts f s

binaryTest :: (Show a, Show b, Show c, Eq c) =>
    [((a,b),c)] -> ( a -> b -> c ) -> String -> Bool
binaryTest ts f s = unaryTest ts ( uncurry f ) s

-- Fuer Funktionen hoeherer Ordnung (Funktionen koennen nicht als
--  Argumente angezeigt werden)
unaryTestNoShowInput :: (Show b, Eq b) =>
    [(a, b)] -> ( a -> b ) -> String -> Bool
unaryTestNoShowInput [] _ _ = True
unaryTestNoShowInput ((x,y):ts) f s =
    if( y /= f x )
    then error ( s ++ ": test failed on unknown input"
        ++ ": expected (" ++ show y ++ "), got ("
        ++ show (f x) ++ ")." )
    else
        unaryTestNoShowInput ts f s

binaryTestNoShowInput :: (Show c, Eq c) =>
    [((a,b),c)] -> ( a -> b -> c ) -> String -> Bool
binaryTestNoShowInput ts f s = unaryTestNoShowInput ts ( uncurry f ) s

ternaryTestNoShowInput :: (Show d, Eq d) =>
    [((a,b,c),d)] -> ( a -> b -> c -> d ) -> String -> Bool
ternaryTestNoShowInput ts f s = unaryTestNoShowInput ts ( uncurry3 f ) s

-- Helferfunktionen zu Allgemeinen Testfunktionen
uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
uncurry3 f (x, y, z) = f x y z

---- Testfunktionen
allTest :: Bool
allTest = ngeradeTest && nmaxTest && nat2IntTest
        && int2NatTest
        && zpowTest && zabsTest && zggtTest && zint2IntTest
        && int2ZintTest
        && fullTest && minDepthTest && heightTest && insertLeafTest
        && insertLeafsTest && deleteLeafTest && deleteLeafsTest
```

```
        && twoChildrenTest && mapTreeTest && foldTreeTest
        && depthTest && sizeTest

ngeradeTest :: Bool
ngeradeTest = unaryTest ngeradeTestData ngerade "ngerade"
nmaxTest :: Bool
nmaxTest = binaryTest nmaxTestData nmax "nmax"
nat2IntTest :: Bool
nat2IntTest = unaryTest nat2IntTestData nat2Int "nat2Int"
int2NatTest :: Bool
int2NatTest = unaryTest int2NatTestData int2Nat "int2Nat"
zpowTest :: Bool
zpowTest = binaryTest zpowTestData zpow "zpow"
zabsTest :: Bool
zabsTest = unaryTest zabsTestData zabs "zabs"
zggtTest :: Bool
zggtTest = binaryTest zggtTestData zggt "zggt"
zint2IntTest :: Bool
zint2IntTest = unaryTest zint2IntTestData zint2Int "zint2Int"
int2ZintTest :: Bool
int2ZintTest = unaryTest int2ZintTestData int2Zint "int2Zint"

fullTest :: Bool
fullTest = unaryTest fullTestData full "full"
minDepthTest :: Bool
minDepthTest = unaryTest minDepthTestData minDepth "minDepth"
heightTest :: Bool
heightTest = unaryTest heightTestData height "height"
insertLeafTest :: Bool
insertLeafTest = unaryTest insertLeafTestData insertLeaf "insertLeaf"
insertLeafsTest :: Bool
insertLeafsTest = binaryTest insertLeafsTestData insertLeafs "insertLeafs"
deleteLeafTest :: Bool
deleteLeafTest = unaryTest deleteLeafTestData deleteLeaf "deleteLeaf"
deleteLeafsTest :: Bool
deleteLeafsTest = binaryTest deleteLeafsTestData deleteLeafs "deleteLeafs"

twoChildrenTest :: Bool
twoChildrenTest = unaryTest twoChildrenTestData twoChildren "twoChildren"
mapTreeTest :: Bool
mapTreeTest = binaryTestNoShowInput mapTreeTestData mapTree "mapTree"
foldTreeTest :: Bool
foldTreeTest = ternaryTestNoShowInput foldTreeTestData foldTree "foldTree"
depthTest :: Bool
depthTest = unaryTest depthTestData depth "depth"
sizeTest :: Bool
sizeTest = unaryTest sizeTestData size "size"
```

From:
http://192.168.10.43/dokuwiki/ - **Knoffhoff**

Permanent link:
http://192.168.10.43/dokuwiki/doku.php/uni/alp1

Last update: **2013/12/02 15:10**