fhwedel UNIVERSITY OF APPLIED SCIENCES

Funktionale Programmierung mit Haskell

Unendliche Listen

Michael Suthe (ms2798@fh-wedel.de)

fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
  - Vergleich zu endlichen Listen
  - Primzahlen
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

Vergleich zu endlichen Listen

Sowohl für endliche, als auch für unendliche Listen:

```
head [n..] = n
take n [1..] = [1..n]
[m..]!!n = m + n
```

List Comprehensions

```
[square x | x <- [1..], odd x]
```

Vergleich zu endlichen Listen

```
> [square x | x <- [0..], square x < 10]
[0,1,2,4,9
```

Equivalenter Ausdruck:

```
filter (< 10) (map square [0..])
```

Mit Vorwissen umformbar:

```
takeWhile (< 10) (map square [0..])
```

fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- **Einleitung**
  - **Vergleich zu endlichen Listen**
  - **Primzahlen**
- **Unendliche Listen als Grenzwerte**
- **Eigenschaften unendlicher Listen**
- **Zyklische Strukturen**
- **Streams**
- **Beispiele**

Primzahlen

Das Sieb des Eratosthenes:

1. Es werden alle Zahlen zwischen 2 und m betrachtet
2. Die Variable n wird auf 2 initialisiert
3. Alle Vielfachen von n im Zahlenfeld werden entfernt
4. n wird auf die nächste verbleibende Zahl gesetzt
5. Schritte 3 und 4 werden solange wiederholt, bis n größer als $\sqrt{m}$ ist

```
primes = sieve [2..m]
    where   sieve (p:xs) = p : sieve [x | x <- xs, not (x `mod` p == 0)]
            sieve [] = []
```

Primzahlen

Für alle Primzahlen:

```
primes = sieve [2..]
    where   sieve (p:xs) = p : sieve [x | x <- xs, not (x `mod` p == 0)]
```

Primzahlen

Derzeit größte bekannte Primzahl:

29941042940415717208904892634044693825736772297541847354767734860009764022
11007410262658651099123208584933441564152126353352134996699849464660024345
64247027257716956426621052611077416379956346589355834130669179364555490042
05895126271181099996307160208959114624960584555225124517504061464679674277
58141698779773518957789226523399152295216195147795568313648450268950958240
52712207416118596253594344535443908358061475952581306252393965564387213568
80887010955400164710207751267172067086114847037838015823014759469842856323
33679380628534371335472004966032794193584306940762774784773227259684214422
13903047497381792744687785378954139863274105174997747578653752738664908983
34413440850589294285939641837847885180293953612316717800454442371278289600
03078763706287784816169234017176752530010427856637688008454993938972775477
95942475140979474097047922818444102160358261198743623792579029732412281401
46132185717900095701932209620447889139768818778378780669150213180115263923
21063980791339465643957490650749674060110378104901912204214147099921817425
03952882198827163727056558548365300076534544813028554202204144649095067903
00106037953557375035573460048013433964321236653564196203053926392735378089
15209945715383919693563659958605226920587103409125566337764968741755986319
97686631333272749910678445871654245230999488724305258809425533012377276026

Michael Suthe (ms2798@fh-wedel.de)

Primzahlen

2062247543939391244474441345784403761363233106401407853243337091275020253
46071522072496259982462272441508118997618168933743055175267793760490172946
23605328701471137559953852941878878356306358329366248552542546493486127574
70131749798366490837792085982491676425443217752003078538280175102296498224
40218590789124546439592257668717470889362098239674999126994976422626666251
61104946651585173982672412936845890484125176102808402126341749691501043856
90034647366375646777823538254026349847948472385331829968989265025765927486
49130305554916313313866289714745469301277164201805138310017941027929338411
27729762994700967129123013969919774514745165689925491609265838453590756819
38652250207646193974393981714275603531026082200843109207824130469888677579
04215092997821807204705338514090035015030239754409536260237719145330983785
53300944012041391408947893906145384786691175424595255535572135346782178589
18572551793172518095269820163050275389257834184338583625866134878869967591
81828824181786086265733473640886059016992162821954578450802518084406868063
55388296318808519569040782914582854478887274874384440248665206587757738011
47177024106733117151260904342633898228185492911478476067166432679353560564
34320008580196058168632037750861631903779201377746365365764114760576405204
82149055194676245740146214915084236912330184647647148651174943460948146868
76112131071814980511406532428781251733105610192591808156657661563000054288
44341832645762955316849646133238129763143299999915781575970850597707514527
05479049588093122344184293631344531763453656641589400154355542210995760935

Primzahlen

378747687134625825164489489475756899117190078680230109764978944314940442 73
917707608160559232395856118484860051345741876849147728619698280088111180 34
515918606652389208306683551868888464279823463653614830668021127763486509 22
981360308146440983106446599002234205636522632004813448661599091974317032 80
998289119167199551902328839851761951646328898636559587788396996874805505 79
066715707455734538315507665530044363555465961917779865072764397412553347 18
67959138518811760133262896968890188027396915...

oder $2^{24.036.583} - 1$ mit 7.235.733 Dezimalstellen

Michael Suthe (ms2798@fh-wedel.de)

Primzahlen

Electronic Frontier Foundation: 100.000 $ für den ersten Primzahlnachweis mit mehr als 10.000.000 Dezimalstellen

Näherung von Carl Friedrich Gauß:

Anzahl der Primzahlen im Bereich von 1 – n entspricht etwa

$$\frac{n}{(\ln(n))}$$

Indizierter Zugriff auf das Element

$$\frac{10^{10000000}}{(\ln(10^{10000000}))}$$

Primzahlen

```
> primes !! (round (10^10000000 / log(10^10000000)))
2

> round (10^10000000 / log(10^10000000))
0

> round (10^308 / log(10^308))
14100470191664029032136531445309121208356245447968606261609809599713611003
73247290997419355553852210963015014010535732060164678096024851549052832470
51405794319033862776806150393493391612978924045777638524694293385125276846
15751053213559162492163130656409988924811689752175495897236886871700873337
5870009344
```

Inhalt

- Einleitung
  - Vergleich zu endlichen Listen
  - Primzahlen
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
  - Grundlagen zu Grenzwerten
  - Ordnungen über Approximationen
  - Berechenbare Funktionen
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

Grundlagen zu Grenzwerten

fhwedel UNIVERSITY OF APPLIED SCIENCES

Mathematik:

Unendlichen Sequenz der Approximationen:

3

3,1

3,14

3,141

3,1415

...

Grenzwert:

π = 3.141592653589...

Grundlagen zu Grenzwerten

```
⊥
1 : ⊥
1 : 2 : ⊥
1 : 2 : 3 : ⊥
1 : 2 : 3 : 4 : ⊥
1 : 2 : 3 : 4 : 5 : ⊥
...
```

```
[1..]
```

Grundlagen zu Grenzwerten

$\bot$

1 : $\bot$

2 : 1 : $\bot$

3 : 2 : 1 : $\bot$

4 : 3 : 2 : 1 : $\bot$

5 : 4 : 3 : 2 : 1 : $\bot$

6 : 5 : 4 : 3 : 2 : 1 : $\bot$

...

Grundlagen zu Grenzwerten

```
⊥
1 : ⊥
1 : 2 : ⊥
1 : 2 : 3 : ⊥
1 : 2 : 3 : ⊥
1 : 2 : 3 : ⊥
...
```

```
1 : 2 : 3 : ⊥
```

Grundlagen zu Grenzwerten

```
approx   :: Integer -> [a] -> [a]
approx (n + 1) [] = []
approx (n + 1) (x : xs) = x : approx n xs
```

```
approx 0 [1] = ⊥
approx 1 [1] = 1 : ⊥
approx 2 [1] = 1 : []
```

lim n→∞ approx n xs = xs

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
  - Grundlagen zu Grenzwerten
  - Ordnungen über Approximationen
  - Berechenbare Funktionen
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

Michael Suthe (ms2798@fh-wedel.de)

Ordnungen über Approximationen

Partielle Ordnung: $\subseteq$ für „ist Approximation von“

Reflexivität:

$x \subseteq x$

Transitivität:

$(x \subseteq y) \wedge (y \subseteq z) \Rightarrow (x \subseteq z)$

Antisymmetrie:

$(x \subseteq y) \wedge (y \subseteq x) \Rightarrow (x = y)$

Ordnungen über Approximationen

Für Approximationsordnungen über Zahlen, Booleans, Character und Aufzählungstypen:

$x \sqsubseteq y \equiv (x = \perp) \vee (x = y)$

Ordnungen über Approximationen

Für Listen:

$\bot \subseteq$ xs

[] $\subseteq$ xs $\equiv$ xs = []

(x : xs) $\subseteq$ (y : ys) $\equiv$ (x $\subseteq$ y) $\wedge$ (xs $\subseteq$ ys)

Ordnungen über Approximationen

$[1, \bot, 3] \sqsubseteq [1, 2, 3]$

$[1, 2, \bot] \sqsubseteq [1, 2, 3]$

fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
  - Grundlagen zu Grenzwerten
  - Ordnungen über Approximationen
  - Berechenbare Funktionen
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

Berechenbare Funktionen

Eine berechenbare Funktion f ist in Bezug auf die Approximationsordung monoton:

Für alle x und y:

$x \subseteq y \Rightarrow f\ x \subseteq f\ y$

Eine berechenbare Funktion f ist stetig:

Für alle Approximationsketten:

$f\ (\lim_{n \to \infty} x_n) = \lim_{n \to \infty} f\ x_n$

Berechenbare Funktionen

```
⊥
1 : ⊥
1 : 2 : ⊥
1 : 2 : 3 : ⊥
1 : 2 : 3 : 4 : ⊥
1 : 2 : 3 : 4 : 5 : ⊥
...

[1..]
```

fhwedel UNIVERSITY OF APPLIED SCIENCES

Berechenbare Funktionen

```
map square (⊥)
map square (1 : ⊥)
map square (1 : 2 : ⊥)
map square (1 : 2 : 3 : ⊥)
map square (1 : 2 : 3 : 4 : ⊥)
map square (1 : 2 : 3 : 4 : 5 : ⊥)
...

map square [1..]
```

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
  - Grundlagen zu Grenzwerten
  - Ordnungen über Approximationen
  - Berechenbare Funktionen
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- **Eigenschaften unendlicher Listen**
- **Zyklische Strukturen**
- **Streams**
- **Beispiele**

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Eigenschaften unendlicher Listen

Problem:

Induktion reicht nicht immer zum Beweis von Eigenschaften unendlicher Listen aus!

Eigenschaften unendlicher Listen


```
iterate  :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
iterate f x = x : map f (iterate f x)
```

Michael Suthe (ms2798@fh-wedel.de)

Eigenschaften unendlicher Listen

Annahme:

```
xs !! n = ys !! n
```

Problem:

```
xs = ⊥
ys = [⊥]

xs !! 0 = ⊥
ys !! 0 = ⊥
```

Eigenschaften unendlicher Listen

```
lim n→∞ approx n xs = xs

approx n xs = approx n ys
```

Zu beweisen:

```
iterate f (f x) = map f (iterate f x)

approx n (iterate f (f x)) = approx n (map f (iterate f x))
```

Eigenschaften unendlicher Listen

Induktionsanfang:

```
approx 0 (iterate f (f x)) = approx 0 (map f (iterate f x))

⊥ = ⊥
```

Eigenschaften unendlicher Listen

Induktionsvorraussetzung:

```
approx n (iterate f (f x)) = approx n (map f (iterate f x))
```

Eigenschaften unendlicher Listen

Induktionsbehauptung:

```
approx (n + 1) (iterate f (f x)) = approx (n + 1) (map f (iterate f x))
```

Eigenschaften unendlicher Listen

Induktionsschritt:

Linke Seite:

```
approx (n + 1) (iterate f (f x))

= approx (n + 1) (f x : iterate f (f (f x)))   -- Definition iterate

= f x : approx n (iterate f (f (f x)))         -- Definition approx

= f x : approx n (map f (iterate f (f x)))     -- Induktionshypothese
```

Eigenschaften unendlicher Listen

Rechte Seite:

```
approx (n + 1) (map f (iterate f x))

= approx (n + 1) (map f (x : iterate f (f x)))     -- Definition iterate

= approx (n + 1) (f x : map f ( iterate f (f x))) -- Definition map

= f x : approx n (map f (iterate f (f x)))         -- Definition approx
```

Eigenschaften unendlicher Listen

Linke Seite:

```
f x : approx n (map f (iterate f (f x)))
```

Rechte Seite:

```
f x : approx n (map f (iterate f (f x)))
```

```
q.e.d
```

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- Beispiele

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- **Zyklische Strukturen**
  - **Grundlagen zyklischer Strukturen**
  - **Hamming-Problem**
- **Streams**
- **Beispiele**

Grundlagen zyklischer Strukturen

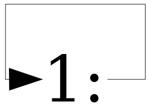
Einfaches Beispiel:

```
ones :: [Int]
ones = 1 : ones
```

Auswertung:

```
ones = 1 : ones
ones = 1 : 1 : ones
ones = 1 : 1 : 1 : ones
...
```

Interne Darstellung der Struktur als Graph:



Grundlagen zyklischer Strukturen

```
more :: String
more = "More" ++ andmore
    where   andmore = "and more" ++ andmore
```

```
> more
More and more and more and more and more and more and more and more and more and
more and more and more and more and more and more and more and more and more and
more and more and more and more and more and more and more and more and more and
more and more and more and more and more and more and more and more and more and
more and more and more and more and more and more and more and more and more and
more and more and more and more and more and more and more and more and more and
...
```

'M':'o':'r':'e':' ': ►'a':'n':'d':' ':'m':'o':'r':'e':

Grundlagen zyklischer Strukturen

```
repeat :: a -> [a]
repeat x = x : reapeat x

ones = repeat 1

> ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
...
```

Grundlagen zyklischer Strukturen

Auswertung:

```
repeat 1
1 : repeat 1
1 : 1 : repeat 1
1 : 1 : 1 : repeat 1
1 : 1 : 1 : 1 : repeat 1
1 : 1 : 1 : 1 : 1 : repeat 1
...
```

Alternative Implemetierung:

```
repeat x = xs
    where   xs = x : xs
```

fhwedel UNIVERSITY OF APPLIED SCIENCES

Grundlagen zyklischer Strukturen

```
iterate f x = x : map f (iterate f x)
```

Auswertung:

```
iterate (2 *) 1
1 : map (2 *) (iterate (2 *) 1)
1 : 2 : map (2 *) (map (2 *) (iterate (2 *) 1))
1 : 2 : 3 : map (2 *) (map (2 *) (map (2 *) (iterate (2 *) 1)))
...
```

Berechenbar in $\Omega(n^2)$ **Schritten**

Grundlagen zyklischer Strukturen


```
iterate f x = xs
    where   xs = x : map f xs
```

Auswertung:

```
1 : map (2 *)
1 : 2 : map (2 *)
1 : 2 : 4 : map (2 *)
1 : 2 : 4 : 8 : map (2 *)
...
```

Berechenbar in $O(\mathrm{n})$ **Schritten**

fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
  - Grundlagen zyklischer Strukturen
  - Hamming-Problem
- Streams
- Beispiele

Hamming-Problem

W. R. Hamming:

Man schreibe ein Programm, das eine Liste mit folgenden Eigenschaften erstellt:

1. Die Liste ist aufsteigend sortiert
2. Die Liste beginnt mit der Zahl 1
3. Wenn die Zah x enthalten ist, so sind auch 2x, 3x und 5x enthalten
4. Die Liste enthält keine anderen Zahlen

Hüllenproblem:

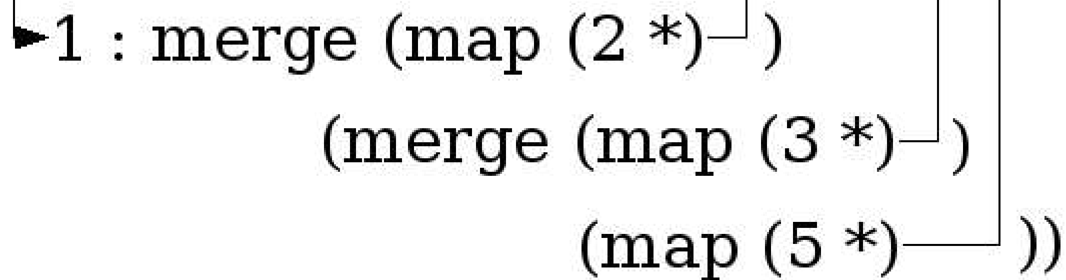
Wendet man auf eine Menge von Initialisierungselementen die generierenden Funktionen an, so erhält man das Ergebnis.

Hamming-Problem

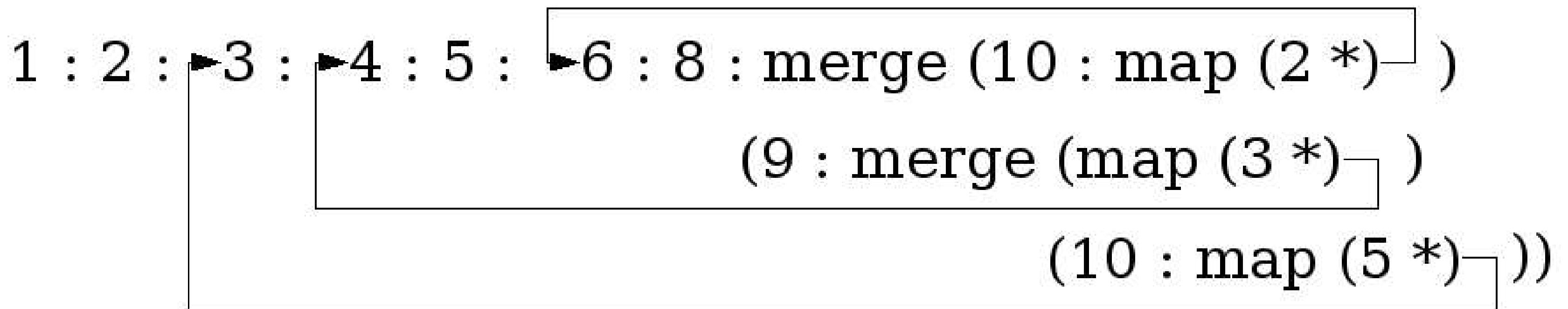
```
merge :: [Integer] -> [Integer] -> [Integer]
merge (x : xs) (y : ys) | x < y = x : merge xs (y : ys)
                        | x == y = x : merge xs ys
                        | x > y = y : merge (x : xs) ys


hamming :: [Integer]
hamming = 1 : merge (map (2 *) hamming)
                    (merge (map (3 *) hamming) (map (5 *) hamming))
```

Hamming-Problem



Hamming-Problem



fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- **Zyklische Strukturen**
  - Grundlagen zyklischer Strukturen
  - Hamming-Problem
- **Streams**
- **Beispiele**

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- **Streams**
- **Beispiele**

fhwedel UNIVERSITY OF APPLIED SCIENCES

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
  - Grundlagen von Streams
  - Echo
- Beispiele

Grundlagen von Streams

stream-based interaction

```
f :: String -> String
```

```
interact :: (String -> String) -> IO()
```

Michael Suthe (ms2798@fh-wedel.de)

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- **Streams**
  - Grundlagen von Streams
  - **Echo**
- **Beispiele**

fhwedel UNIVERSITY OF APPLIED SCIENCES

Echo

```
capitalise :: Char -> Char
capitalise c = if isLower c then chr (offset + ord c) else c
    where   offset = ord 'A' - ord 'a'
```

```
> interact (map capitalise)
```

Eingabe:

```
introduction into functional programming using haskell
```

Ausgabe:

```
INTRODUCTION INTO FUNCTIONAL PROGRAMMING USING HASKELL
```

Echo

```
> interact ((takeWhile (/= '.')) . (map capitalise))
```

Eingabe:

```
introduction into functional programming using haskell.
```

Ausgabe:

```
INTRODUCTION INTO FUNCTIONAL PROGRAMMING USING HASKELL
```

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- **Streams**
  - Grundlagen von Streams
  - Echo
- **Beispiele**

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- **Beispiele**

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- **Beispiele**
  - **Vigenère-Verschlüsselung**
  - **Papier-Stein-Schere-Spiel**

Vigenère-Verschlüsselung

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Polyalphabetische Verschlüsselung

Vigenère Quadrat

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
IJKLMNOPQRSTU...
```

Vigenère-Verschlüsselung

```
eChar :: Char -> Char -> Char
eChar a b = chr (((ord a - ord 'a' + ord b - ord 'a') `mod` 26) + ord 'a')

dChar :: Char -> Char -> Char
dChar a b = chr (((ord a + 26 - ord b) `mod` 26) + ord 'a')
```

Vigenère-Verschlüsselung

```
key :: String -> String
key a = a ++ key a
```

Michael Suthe (ms2798@fh-wedel.de)

Vigenère-Verschlüsselung

```
eString :: String -> String -> String
eString a b = zipWith eChar a (key b)

dString :: String -> String -> String
dString a b = zipWith dChar a (key b)
```

Vigenère-Verschlüsselung

```
> eString "introductionintofunctionalprogrammingusinghaskell" "hugs"
"phzjvxauacufphzgmotuacufhfvjvaxstgofnoyauanszekds"

> dString "phzjvxauacufphzgmotuacufhfvjvaxstgofnoyauanszekds" "hugs"
"introductionintofunctionalprogrammingusinghaskell"
```

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- **Beispiele**
  - Vigenère-Verschlüsselung
  - **Papier-Stein-Schere-Spiel**

Papier-Stein-Schere-Spiel

Symbole:

- Stein
- Schere
- Papier

Regeln:

- Papier wickelt Stein ein
- Stein stumpft Schere ab
- Schere schneidet Papier

fhwedel UNIVERSITY OF APPLIED SCIENCES

Papier-Stein-Schere-Spiel

```
data Move = Paper | Rock | Scissors

type Round = (Move,Move)

score :: Round -> (Int,Int)
score (x,y) | x beats y = (1,0)
            | y beats x = (0,1)
            | otherwise = (0,0)
```

Michael Suthe (ms2798@fh-wedel.de)

Papier-Stein-Schere-Spiel

```
(beats) :: Move -> Move -> Bool
 x beats y = (m + 1 == n) or (m == n + 2)
    where    m = code x
             n = code y
```

```
code :: Move -> Int
code Paper    = 0
code Rock     = 1
code Scissors = 2
```

Papier-Stein-Schere-Spiel

Strategien:

Möglichkeit 1:

```
type Strategy = [Move] -> Move
```

```
recip :: Strategy
recip ms = if null ms then Rock else last ms
```

Papier-Stein-Schere-Spiel

```
smart :: Strategy
smart ms = if null ms then Rock else choose (count ms)


count :: [Move] -> (Int, Int, Int)
count = foldl (prs) (0, 0, 0)


(prs) :: (Int, Int, Int) -> Move -> (Int, Int, Int)
(p, r, s) prs Paper    = (p + 1, r, s)
(p, r, s) prs Rock     = (p, r + 1, s)
(p, r, s) prs Scissors = (p, r, s + 1)
```

Papier-Stein-Schere-Spiel

```
choose :: (Int,Int,Int) -> Move
choose (p, r, s) | m < p = Scissor
                 | m < p + r = Paper
                 | otherwise = Rock
    where   m = random (p + r + s)
```

Papier-Stein-Schere-Spiel

```
rounds :: (Strategy, Strategy) -> [Round]
rounds (f, g) = (map last . tail . iterate (extend (f, g))) []

extend :: (Strategy, Strategy) -> [Round] -> [Round]
extend (f, g) rs = rs ++ [(f (map snd rs), g (map fst rs))]
```

Papier-Stein-Schere-Spiel

fhwedel UNIVERSITY OF APPLIED SCIENCES

Möglichkeit 2:

```
type strategy = [Move] -> [Move]


type recip ms = Rock : ms


smart xs = Rock : map choose (counts xs)
    where   counts = tail . scanl (prs) (0, 0, 0)
```

Papier-Stein-Schere-Spiel

```
rounds (f, g) = zip xs ys
    where   xs = f ys
            ys = g xs
```

Papier-Stein-Schere-Spiel

```
match :: Int -> (Strategy, Strategy) -> (Int, Int)
match n = total . map score . take n . rounds

total :: [(Int, Int)] -> (Int, Int)
total = pair (sum . map fst, sum . map snd)
```

Papier-Stein-Schere-Spiel

Aber:

```
rounds (f, g) = zip xs ys
    where   xs = f ys
            ys = g xs
```

Der Zugriff auf die Züge des Gegners ermöglicht das Schummeln

```
cheat xs = map trumps xs

trumps :: Move -> Move
trumps Paper    = Scissors
trumps Rock     = Paper
trumps Scissors = Rock
```

Papier-Stein-Schere-Spiel

```
recip [] = [Rock]
cheat [] = []

recip [] = [Rock]
cheat [Rock] = [Paper]

recip [Paper] = [Rock,Paper]
cheat [Rock] = [Paper]

recip [Paper] = [Rock,Paper]
cheat [Rock,Paper] = [Paper,Scissors]
```

Papier-Stein-Schere-Spiel

fhwedel UNIVERSITY OF APPLIED SCIENCES

```
oneshot xs = trumps (head xs) : recip (tail xs)
```

```
devious = take 2 (recip xs) ++ cheat (drop 2 xs)
```

Michael Suthe (ms2798@fh-wedel.de)

Papier-Stein-Schere-Spiel

```
police f xs = ys where ys = f (synch xs ys)

synch :: [Move] -> [Move] -> [Move]
synch (x : xs) (y : ys) = if defined y then x : synch xs ys else ⊥

defined :: Move -> Bool
defined | ⊥ = False
        | otherwise = True



rounds (f,g) = zip xs ys
    where    xs = police f ys
             ys = police g xs
```

Inhalt

- Einleitung
- Unendliche Listen als Grenzwerte
- Eigenschaften unendlicher Listen
- Zyklische Strukturen
- Streams
- **Beispiele**
  - Vigenère-Verschlüsselung
  - Papier-Stein-Schere-Spiel

fhwedel
UNIVERSITY OF APPLIED SCIENCES

Inhalt

- **Einleitung**
- **Unendliche Listen als Grenzwerte**
- **Eigenschaften unendlicher Listen**
- **Zyklische Strukturen**
- **Streams**
- **Beispiele**

Ende

Vielen Dank für Ihre Aufmerksamkeit!