Countdown!

8 out of 10 cats

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

2. November, 2023

List Comprehensions

- *List comprehensions* sind syntaktischer Zucker
- und ähneln stark der Mengenschreibweise in der Mathematik
- Generatoren: Eingabemengen die verarbeitet werden sollen
- Prädikate: Einschränkung der auszugebenden Variablen

$$S = \{(x, y, x^2y^2) | x \in \{1, \ldots, 3\}, x \bmod 2 = 1, y \in \{2, \ldots, 3\}\}$$

```
1  -- [(1,2,4),(1,3,9),(3,2,36),(3,3,81)]
2  compr :: [(Int,Int,Int)]
```

List Comprehensions

- *List comprehensions* sind syntaktischer Zucker
- und ähneln stark der Mengenschreibweise in der Mathematik
- Generatoren: Eingabemengen die verarbeitet werden sollen
- Prädikate: Einschränkung der auszugebenden Variablen

$$S = \{(x, y, x^2y^2) | x \in \{1, \ldots, 3\}, x \bmod 2 = 1, y \in \{2, \ldots, 3\}\}$$

```
1  -- [(1,2,4),(1,3,9),(3,2,36),(3,3,81)]
2  compr :: [(Int,Int,Int)]

3  compr = [ (x,y,x^2*y^2) | x <- [1..3], x `mod` 2 == 1
4                                  -- Generator,  Praedikat
5          , y <- [2..3] ]
```

Generatoren

- Generatoren: `x <- xs`, beschreibt wie x generiert werden
  `[2*x |x <- [1..3]] == [2,4,6]`
- mehrere Generatoren können mit verschachtelten Schleifen verglichen werden `[(x,y)| x<-[1..3], y<-"ABC"] == [(1,'A'),(1,'B'),(1,'C'),(2,'A'),(2,'B'),(2,'C'),(3,'A'),(3,'B'),(3,'C')]`
- Reihenfolge ist relevant: `x <- xs, y <- ys` anders als `y <- ys, x <- xs`
  `[(x,y)|y<-"ABC",x<-[1..3]] == [(1,'A'),(2,'A'),(3,'A'),(1,'B'),(2,'B'),(3,'B'),(1,'C'),(2,'C'),(3,'C')]`
- Abhängigkeiten nach links erlaubt: `x <- xs, y <- [x .. 10]`
  `[(x,y)| x <- [1..3], y <- [x..4]] == [(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4)]`

Prädikate

- Prädikate filtern Kandidaten aus, dabei können alle links stehenden Generatorelemente benutzt werden
- auch bekannt als *guards*
- die Funktionen evaluieren jeweils zu `Bool`

```
1  factors :: Integer -> [Integer]
2  factors p = [ f | f <- [1..p], p `mod` f == 0 ]
3
4  isPrime :: Integer -> Bool
5  isPrime p = [1,p] == factors p
6
7  primes :: [Integer]
8  primes = [ p | p <- [2..], isPrime p ]
```

Primes verbessern?

Was ist an obiger Funktion schlecht und wie könnte eine Verbesserung aussehen?

```
19  primes2 :: [Integer]
20  primes2 = sieve [2..]
```

Frage zum Abschluss hier: ist das zufriedenstellend?

Primes verbessern?

Was ist an obiger Funktion schlecht und wie könnte eine Verbesserung aussehen?

```
19  primes2 :: [Integer]
20  primes2 = sieve [2..]

22  sieve :: [Integer] -> [Integer]
```

Frage zum Abschluss hier: ist das zufriedenstellend?

Primes verbessern?

Was ist an obiger Funktion schlecht und wie könnte eine Verbesserung aussehen?

```
19  primes2 :: [Integer]
20  primes2 = sieve [2..]

22  sieve :: [Integer] -> [Integer]

23  sieve (p:xs) = p
```

Frage zum Abschluss hier: ist das zufriedenstellend?

Primes verbessern?

Was ist an obiger Funktion schlecht und wie könnte eine Verbesserung aussehen?

```
19  primes2 :: [Integer]
20  primes2 = sieve [2..]

22  sieve :: [Integer] -> [Integer]

23  sieve (p:xs) = p

24                 : sieve [ x
```

Frage zum Abschluss hier: ist das zufriedenstellend?

Primes verbessern?

Was ist an obiger Funktion schlecht und wie könnte eine Verbesserung aussehen?

```
19  primes2 :: [Integer]
20  primes2 = sieve [2..]

22  sieve :: [Integer] -> [Integer]

23  sieve (p:xs) = p

24                 : sieve [ x

25                           | x <- xs
```

Frage zum Abschluss hier: ist das zufriedenstellend?

Primes verbessern?

Was ist an obiger Funktion schlecht und wie könnte eine Verbesserung aussehen?

```
19  primes2 :: [Integer]
20  primes2 = sieve [2..]

22  sieve :: [Integer] -> [Integer]

23  sieve (p:xs) = p

24                 : sieve [ x

25                           | x <- xs

26                           , x `mod` p > 0 ]
```

Frage zum Abschluss hier: ist das zufriedenstellend?

Fib and Fib

```
 6  fib :: M.Map Integer Integer
 7  fib = M.fromList ((0,0) : (1,1) : [(k,go k) | k <-
        [2..]])
 8    where
 9      go :: Integer -> Integer
10      go k = fib M.! (k-1) + fib M.! (k-2)
```

Probiert diesen Code einmal aus (oder wir schauen uns an was passiert)

Fib and Fib

```
 6  fib :: M.Map Integer Integer
 7  fib = M.fromList ((0,0) : (1,1) : [(k,go k) | k <-
        [2..]])
 8    where
 9      go :: Integer -> Integer
10      go k = fib M.! (k-1) + fib M.! (k-2)
```

Probiert diesen Code einmal aus (oder wir schauen uns an was passiert)

```
21  fibl :: [Integer]
22  fibl = 0 : 1 : [ go k | k <- [2..] ]
23    where
24      go k = fibl !! (k-1) + fibl !! (k-2)
```

Fib and Fib

```
12  fib' :: Integer -> Integer
13  fib' z = fs M.! z
14    where
15      fs = M.fromList [ (k,v) | k <- [0..z], let v = go k ]
16      go :: Integer -> Integer
17      go 0 = 0
18      go 1 = 1
19      go k = fs M.! (k-1) + fs M.! (k-2)
```

Und im Vergleich

```
21  fibl :: [Integer]
22  fibl = 0 : 1 : [ go k | k <- [2..] ]
23    where
24      go k = fibl !! (k-1) + fibl !! (k-2)
```

Erzeugen aller nichtleeren Teillisten einer Liste

Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
...das heisst in beliebiger Reihenfolge, oder?

```
1  -- | All moeglichen Teilsequenzen einer Liste
2
3  subseqs :: [a] -> [[a]]
```

Erzeugen aller nichtleeren Teillisten einer Liste

Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
...das heisst in beliebiger Reihenfolge, oder?

```
1  -- | All moeglichen Teilsequenzen einer Liste
2
3  subseqs :: [a] -> [[a]]

4  subseqs [x] = [[x]]
```

Erzeugen aller nichtleeren Teillisten einer Liste

Kombiniere bis zu 6 der Zahlen in einem legalen arithmetischen Ausdruck
...das heisst in beliebiger Reihenfolge, oder?

```
1  -- | All moeglichen Teilsequenzen einer Liste
2
3  subseqs :: [a] -> [[a]]

4  subseqs [x] = [[x]]

5  subseqs (x:xs) = xss ++ [x] : map (x:) xss
6    where xss = subseqs xs
```

Nicht-leeres Splitten einer Liste

```
1  -- | Given ordered @xs@, create two ordered sublists,
       which when merged are equal to @xs@.
2  --
3  -- let (ys,zs) = unmerges xs
4  -- forall y in ys . z in zs : merge y z == xs
5
6  unmerges :: Show a => [a] -> [([a],[a])]
```

Nicht-leeres Splitten einer Liste

```
1  -- | Given ordered @xs@, create two ordered sublists,
       which when merged are equal to @xs@.
2  --
3  -- let (ys,zs) = unmerges xs
4  -- forall y in ys . z in zs : merge y z == xs
5
6  unmerges :: Show a => [a] -> [([a],[a])]
```

```
1  unmerges [x,y] = [([x],[y]), ([y],[x])]
```

Nicht-leeres Splitten einer Liste

```
1  -- | Given ordered @xs@, create two ordered sublists,
       which when merged are equal to @xs@.
2  --
3  -- let (ys,zs) = unmerges xs
4  -- forall y in ys . z in zs : merge y z == xs
5
6  unmerges :: Show a => [a] -> [([a],[a])]
```

```
1  unmerges [x,y] = [([x],[y]), ([y],[x])]
```

```
1  unmerges (x:xs) = [([x],xs),(xs,[x])] ++ concatMap (add x
       ) (unmerges xs)
2    where add x (ys,zs) = [(x:ys,zs),(ys,x:zs)]
3  unmerges burn = error $ show burn
```

Einfügen der `Op`

```
1  -- | Combine two expression trees with all possible,
       legal operations
2
3  combine :: (Expr,Int) -> (Expr,Int) -> [(Expr,Int)]
4  combine (l,v) (r,w) = [(App op l r, appOp op v w) | op <-
       ops, legal op v w]
5    where ops = [Add,Sub,Mul,Div] -- [minBound..maxBound]
```

Erstellen aller legalen Expr-Trees

```
1  -- | Generate all possible expressions for our set of
       numbers
2
3  mkExprs :: [Int] -> [(Expr, Int)]
4  mkExprs [x] = [(Zahl x, x)]
5  mkExprs xs = [ ev | (ys,zs) <- unmerges xs
6                    , l <- mkExprs ys, r <- mkExprs zs
7                    , ev <- combine l r ]
```

Loesungen testen

Welche Expr ist am naechsten an "nearest 831" dran?

```
1  nearest :: Int -> [(Expr,Int)] -> (Expr,Int)
2  nearest n ((e,v):evs)
3  -- direkt eine Loesung gefunden?
```

Loesungen testen

Welche Expr ist am naechsten an "nearest 831" dran?

```
1  nearest :: Int -> [(Expr,Int)] -> (Expr,Int)
2  nearest n ((e,v):evs)
3  -- direkt eine Loesung gefunden?
```

```
1     | d == 0 = (e,v)
```

```
1     where d = abs (n-v)
```

Loesungen testen

Welche Expr ist am naechsten an "nearest 831" dran?

```
1  nearest :: Int -> [(Expr,Int)] -> (Expr,Int)
2  nearest n ((e,v):evs)
3  -- direkt eine Loesung gefunden?

1     | d == 0 = (e,v)

1  -- nein? Suche starten, mit Abstand d



1     where d = abs (n-v)
```

Loesungen testen

Welche Expr ist am naechsten an "nearest 831" dran?

```
1  nearest :: Int -> [(Expr,Int)] -> (Expr,Int)
2  nearest n ((e,v):evs)
3  -- direkt eine Loesung gefunden?

1    | d == 0 = (e,v)

1  -- nein? Suche starten, mit Abstand d

1    | otherwise = search n d (e,v) evs

1    where d = abs (n-v)
```

Loesungen testen

Suche solange nach weiteren Loesungen bis die Entfernung 0 ist oder keine weiteren Lsg existieren

```
1  search :: Int -> Int -> (Expr,Int) -> [(Expr,Int)] -> (
       Expr,Int)
```

Loesungen testen

Suche solange nach weiteren Loesungen bis die Entfernung 0 ist oder keine weiteren Lsg existieren

```
1  search :: Int -> Int -> (Expr,Int) -> [(Expr,Int)] -> (
       Expr,Int)
```

```
1  -- es gibt nur suboptimale Lsg
2  search n d ev [] = ev
```

```
1    where d' = abs (n-v)
```

Loesungen testen

Suche solange nach weiteren Loesungen bis die Entfernung 0 ist oder keine weiteren Lsg existieren

```
1  search :: Int -> Int -> (Expr,Int) -> [(Expr,Int)] -> (
       Expr,Int)
```

```
1  -- es gibt nur suboptimale Lsg
2  search n d ev [] = ev
```

```
1  search n d ev ((e,v):evs)
2  -- optimal
3    | d' == 0 = (e,v)
4  -- besser
5    | d' < d  = search n d' (e,v) evs
6  -- schlechter
7    | d' >= d = search n d ev evs
```

```
1    where d' = abs (n-v)
```