

Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

- 1 Double-Linked List
- 2 Haskell-Lösungen
- 3 Laziness
- 4 Fazit: Laziness

Worum soll es heute gehen?

- **Funktionale Programmierung** generell

Worum soll es heute gehen?

- **Funktionale Programmierung** generell
- Implementierung einer **Double-Linked-List**
Wie macht man sowas in Haskell?

Worum soll es heute gehen?

- **Funktionale Programmierung** generell
- Implementierung einer **Double-Linked-List**
Wie macht man sowas in Haskell?
- **Laziness**
Was für Auswirkungen hat das auf die Programmierung?
Was für Möglichkeiten bietet dies?

Eine Double-Linked-List ist *die* klassische Einstiegsdatenstruktur in der imperativen Welt.

Eine Double-Linked-List ist *die* klassische Einstiegsdatenstruktur in der imperativen Welt.

Sie besteht aus...

- einem Paar von (äußeren) Pointern, die auf den Anfang und das Ende zeigen und

Eine Double-Linked-List ist *die* klassische Einstiegsdatenstruktur in der imperativen Welt.

Sie besteht aus. . .

- einem Paar von (äußeren) Pointern, die auf den Anfang und das Ende zeigen und
- Elementen, wiederum zusammengesetzt aus:

Eine Double-Linked-List ist *die* klassische Einstiegsdatenstruktur in der imperativen Welt.

Sie besteht aus...

- einem Paar von (äußeren) Pointern, die auf den Anfang und das Ende zeigen und
- Elementen, wiederum zusammengesetzt aus:
 - Einem Pointer auf das nächste Element (null falls Ende der Liste)

Eine Double-Linked-List ist *die* klassische Einstiegsdatenstruktur in der imperativen Welt.

Sie besteht aus...

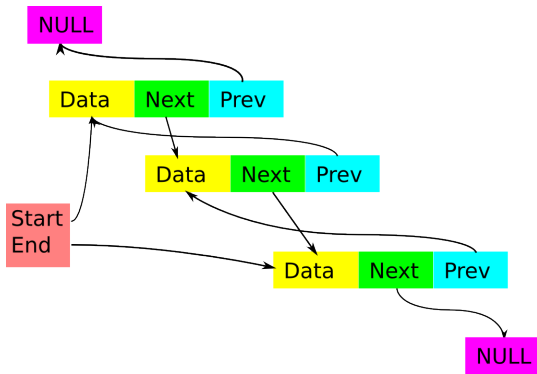
- einem Paar von (äußeren) Pointern, die auf den Anfang und das Ende zeigen und
- Elementen, wiederum zusammengesetzt aus:
 - Einem Pointer auf das nächste Element (`null` falls Ende der Liste)
 - Einem Pointer auf das vorherige Element (`null` falls Anfang der Liste)

Eine Double-Linked-List ist *die* klassische Einstiegsdatenstruktur in der imperativen Welt.

Sie besteht aus...

- einem Paar von (äußeren) Pointern, die auf den Anfang und das Ende zeigen und
- Elementen, wiederum zusammengesetzt aus:
 - Einem Pointer auf das nächste Element (`null` falls Ende der Liste)
 - Einem Pointer auf das vorherige Element (`null` falls Anfang der Liste)
 - Einem Datum, welches gespeichert werden soll (beliebig)

DLL als Grafik:



Die Vorteile dieser Datenstruktur liegen auf der Hand:

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (cons/snoc) liegt in $\mathcal{O}(1)$.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (cons/snoc) liegt in $\mathcal{O}(1)$.
- Iteration (z.B. `map`) ist einfach zu implementieren.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (cons/snoc) liegt in $\mathcal{O}(1)$.
- Iteration (z.B. `map`) ist einfach zu implementieren.
- Finden/Updaten eines Elementes liegt in $\mathcal{O}(n)$.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (cons/snoc) liegt in $\mathcal{O}(1)$.
- Iteration (z.B. `map`) ist einfach zu implementieren.
- Finden/Updaten eines Elementes liegt in $\mathcal{O}(n)$.
- Verbinden von 2 Listen liegt in $\mathcal{O}(1)$.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (cons/snoc) liegt in $\mathcal{O}(1)$.
- Iteration (z.B. map) ist einfach zu implementieren.
- Finden/Updaten eines Elementes liegt in $\mathcal{O}(n)$.
- Verbinden von 2 Listen liegt in $\mathcal{O}(1)$.

Die einfache Liste in Haskell hat auch diese Eigenschaften - allerdings nur am vorderen Ende.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (cons/snoc) liegt in $\mathcal{O}(1)$.
- Iteration (z.B. map) ist einfach zu implementieren.
- Finden/Updaten eines Elementes liegt in $\mathcal{O}(n)$.
- Verbinden von 2 Listen liegt in $\mathcal{O}(1)$.

Die einfache Liste in Haskell hat auch diese Eigenschaften - allerdings nur am vorderen Ende.

Einfügen von hinten und Verkettung läuft immernoch in $\mathcal{O}(n)$.

Die Vorteile dieser Datenstruktur liegen auf der Hand:

- Einf. eines Elementes vorne/hinten (`cons/snoc`) liegt in $\mathcal{O}(1)$.
- Iteration (z.B. `map`) ist einfach zu implementieren.
- Finden/Updaten eines Elementes liegt in $\mathcal{O}(n)$.
- Verbinden von 2 Listen liegt in $\mathcal{O}(1)$.

Die einfache Liste in Haskell hat auch diese Eigenschaften - allerdings nur am vorderen Ende.

Einfügen von hinten und Verkettung läuft immernoch in $\mathcal{O}(n)$.

Wie bekommen wir nun alle Vorteile nach Haskell-Land? Und wieso gibt es da nichts in der Prelude?

Also implementieren wir einfach mal diese Datenstruktur:

Also implementieren wir einfach mal diese Datenstruktur:

Einen Pointer in Haskell bekommen und bearbeiten wir mittels

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Also implementieren wir einfach mal diese Datenstruktur:

Einen Pointer in Haskell bekommen und bearbeiten wir mittels

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Eine Doubly-Linked-List ist dann:

```
data Entry a = Entry { prev :: IORef (Maybe (Entry a))
                      , data :: a
                      , next :: IORef (Maybe (Entry a))
                      }
data Dll a    = Dll   { first :: IORef (Maybe (Entry a))
                      , last  :: IORef (Maybe (Entry a))
                      }
```

Also implementieren wir einfach mal diese Datenstruktur:

Einen Pointer in Haskell bekommen und bearbeiten wir mittels

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

Eine Doubly-Linked-List ist dann:

```
data Entry a = Entry { prev :: IORef (Maybe (Entry a))
                      , data :: a
                      , next :: IORef (Maybe (Entry a))
                      }
data Dll a    = Dll   { first :: IORef (Maybe (Entry a))
                      , last  :: IORef (Maybe (Entry a))
                      }
```

wobei das Maybe-Konstrukt im Entry einmal den Pointer zurück und einmal den Pointer nach vorn kennzeichnet und das Maybe-Konstrukt im Dll eine leere Liste ermöglicht.

Wir können dies nun runter implementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (D11 a)
insertFront :: a -> D11 a -> IO (D11 a)
insertBack  :: a -> D11 a -> IO (D11 a)
getFront    :: D11 a -> IO (Maybe a)
getBack     :: D11 a -> IO (Maybe a)
map         :: (a -> b) -> D11 a -> IO (D11 b)
...

```

Wir können dies nun runter implementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (D11 a)
insertFront :: a -> D11 a -> IO (D11 a)
insertBack  :: a -> D11 a -> IO (D11 a)
getFront    :: D11 a -> IO (Maybe a)
getBack     :: D11 a -> IO (Maybe a)
map         :: (a -> b) -> D11 a -> IO (D11 b)
...

```

Probleme:

- Alles ist in IO (wegen IORefs).

Wir können dies nun runter implementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (D11 a)
insertFront :: a -> D11 a -> IO (D11 a)
insertBack  :: a -> D11 a -> IO (D11 a)
getFront    :: D11 a -> IO (Maybe a)
getBack     :: D11 a -> IO (Maybe a)
map         :: (a -> b) -> D11 a -> IO (D11 b)
...

```

Probleme:

- Alles ist in IO (wegen IORefs).
- Wir können es aufgrund von IO in keinem reinen Code benutzen.

Wir können dies nun runter implementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (D11 a)
insertFront :: a -> D11 a -> IO (D11 a)
insertBack  :: a -> D11 a -> IO (D11 a)
getFront    :: D11 a -> IO (Maybe a)
getBack     :: D11 a -> IO (Maybe a)
map         :: (a -> b) -> D11 a -> IO (D11 b)
...

```

Probleme:

- Alles ist in IO (wegen IORefs).
- Wir können es aufgrund von IO in keinem puren Code benutzen.
- Wer garantiert uns, dass die Struktur nur Daten hält und (ggf. nach einem „Patch“) nicht per IO Raketen abschießt?

Wir können dies nun runter implementieren und erhalten ein Interface ähnlich zu

```
newDLL      :: IO (D11 a)
insertFront :: a -> D11 a -> IO (D11 a)
insertBack  :: a -> D11 a -> IO (D11 a)
getFront    :: D11 a -> IO (Maybe a)
getBack     :: D11 a -> IO (Maybe a)
map         :: (a -> b) -> D11 a -> IO (D11 b)
...

```

Probleme:

- Alles ist in IO (wegen IORefs).
- Wir können es aufgrund von IO in keinem puren Code benutzen.
- Wer garantiert uns, dass die Struktur nur Daten hält und (ggf. nach einem „Patch“) nicht per IO Raketen abschießt?

Das ist keine gute Lösung! Vor allem nicht funktional!

Nochmal Vor- & Nachteile:

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
 - Zusätzliche Pointer im Maybe

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
 - Zusätzliche Pointer im Maybe
 - Zusätzliche Pointer durch das IORef

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
 - Zusätzliche Pointer im Maybe
 - Zusätzliche Pointer durch das IORef
 - statt drei Pointer (Datum, prev und next) speichern wir sieben (1x prev, 1x next, 2x Maybe, 2x IORef, 1x Datum)

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
 - Zusätzliche Pointer im Maybe
 - Zusätzliche Pointer durch das IORef
 - statt drei Pointer (Datum, prev und next) speichern wir sieben (1x prev, 1x next, 2x Maybe, 2x IORef, 1x Datum)
 - „Pointer jagen“ im Speicher kostet zusätzliche Zeit beim Lookup

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
 - Zusätzliche Pointer im Maybe
 - Zusätzliche Pointer durch das IORef
 - statt drei Pointer (Datum, prev und next) speichern wir sieben (1x prev, 1x next, 2x Maybe, 2x IORef, 1x Datum)
 - „Pointer jagen“ im Speicher kostet zusätzliche Zeit beim Lookup

Insgesamt ist es keine gute Idee die Datenstrukturen aus der imperativen Programmierung einfach nachzubauen.

Nochmal Vor- & Nachteile:

Vorteile:

- Von den Zugriffszeiten genau das, was wir wollten

Nachteile:

- praktisch Unbrauchbar durch IO
- VIEL Speicheraufwändiger als die C-Lösung, durch
 - Zusätzliche Pointer im Maybe
 - Zusätzliche Pointer durch das IORef
 - statt drei Pointer (Datum, prev und next) speichern wir sieben (1x prev, 1x next, 2x Maybe, 2x IORef, 1x Datum)
 - „Pointer jagen“ im Speicher kostet zusätzliche Zeit beim Lookup

Insgesamt ist es keine gute Idee die Datenstrukturen aus der imperativen Programmierung einfach nachzubauen.

Aber wie geht es dann?

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (i.e. `map`) über dieses Ding

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (i.e. `map`) über dieses Ding
- Möglichkeit, Elemente aus der Mitte zu entfernen

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (i.e. `map`) über dieses Ding
- Möglichkeit, Elemente aus der Mitte zu entfernen

und weil wir schonmal dabei sind:

- immutable und pure

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (i.e. `map`) über dieses Ding
- Möglichkeit, Elemente aus der Mitte zu entfernen

und weil wir schonmal dabei sind:

- immutable und pure
- Wenig Speicherverbrauch

Wir fangen einfach mal an mit einem Wunschkonzert. Wir hätten gerne:

- Ein Sequence-Ähnliches Ding (Array, Liste, etc.)
- Schnelles Hinzufügen/Entfernen von Elementen am Anfang/Ende
- Schnelles Zusammenfügen zweier dieser Dinge
- Iterieren (i.e. map) über dieses Ding
- Möglichkeit, Elemente aus der Mitte zu entfernen

und weil wir schonmal dabei sind:

- immutable und pure
- Wenig Speicherverbrauch

Die erste Liste ist die typische Problemstellung deren Lösung eine Doubly-Linked-List ist - mit den Zusatzanforderungen müssen wir uns was anderes überlegen.

Wie könnte so eine API (minimal) aussehen?

Wie könnte so eine API (minimal) aussehen?

```
data Ding a = ... -- implementation goes here
```

```
empty    :: Ding a  
isEmpty  :: Ding a -> Bool  
append   :: Ding a -> a -> Ding a  
prepend  :: Ding a -> a -> Ding a  
getFirst :: Ding a -> Maybe (a, Ding a)  
getLast  :: Ding a -> Maybe (a, Ding a)  
concat   :: Ding a -> Ding a -> Ding a
```

Wenn das alles ist, dann können wir das auch zu einer Typklasse machen

Wenn das alles ist, dann können wir das auch zu einer Typklasse machen

```
class Deque d where
  empty      :: d a
  isEmpty    :: d a -> Bool
  append     :: d a -> a -> d a
  prepend    :: d a -> a -> d a
  getFirst   :: d a -> Maybe (a, d a)
  getLast    :: d a -> Maybe (a, d a)
  concat     :: d a -> d a -> d a
```

Mögliche Lösung: Zwei Listen

```
data TwoLists a = TwoLists { front :: [a]  
                             , back  :: [a]  
                             }
```

Mögliche Lösung: Zwei Listen

```
data TwoLists a = TwoLists { front :: [a]  
                             , back  :: [a]  
                             }
```

Wie implementieren wir jetzt die API?

API (cont.)

```
empty = TwoLists [] []
```

API (cont.)

```
empty = TwoLists [] []
```

```
isEmpty (TwoLists [] []) = True  
isEmpty _                 = False
```


API (cont.)

```
empty = TwoLists [] []
```

```
isEmpty (TwoLists [] []) = True  
isEmpty _                 = False
```

```
prepend a (TwoLists front back) = TwoLists (a:front) back
```

API (cont.)

```
empty = TwoLists [] []
```

```
isEmpty (TwoLists [] []) = True  
isEmpty _                 = False
```

```
prepend a (TwoLists front back) = TwoLists (a:front) back
```

```
append a (TwoLists front back) = TwoLists front (a:back)
```

```
getFirst (TwoLists [] []) = Nothing
getFirst (TwoLists (a:as) bs) = Just (a, TwoLists as bs)
getFirst (TwoLists [] bs) = let (c:cs) = reverse bs in
                             Just (c, TwoLists cs [])
```

```
getFirst (TwoLists [] []) = Nothing
getFirst (TwoLists (a:as) bs) = Just (a, TwoLists as bs)
getFirst (TwoLists [] bs) = let (c:cs) = reverse bs in
                             Just (c, TwoLists cs [])

getLast (TwoLists [] []) = Nothing
getLast (TwoLists as (b:bs)) = Just (b, TwoLists as bs)
getLast (TwoLists as []) = let (c:cs) = reverse as in
                             Just (c, TwoLists [] cs)
```

```
getFirst (TwoLists []      []) = Nothing
getFirst (TwoLists (a:as) bs) = Just (a, TwoLists as bs)
getFirst (TwoLists []      bs) = let (c:cs) = reverse bs in
                                   Just (c, TwoLists cs [])

getLast  (TwoLists []      []) = Nothing
getLast  (TwoLists as (b:bs)) = Just (b, TwoLists as bs)
getLast  (TwoLists as      []) = let (c:cs) = reverse as in
                                   Just (c, TwoLists [] cs)

concat (TwoLists as bs) (TwoLists cs ds) =
    TwoLists (as ++ reverse bs) (ds ++ reverse cs)
```

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.
Dies zieht alle Operationen, die dies benutzen auf $\mathcal{O}(n)$ hoch.

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.
Dies zieht alle Operationen, die dies benutzen auf $\mathcal{O}(n)$ hoch.

Asymptoten:

- `empty` $\mathcal{O}(1)$
- `append` $\mathcal{O}(1)$
- `prepend` $\mathcal{O}(1)$
- `getFirst` $\mathcal{O}(n)$
- `getLast` $\mathcal{O}(n)$
- `iterate` $\mathcal{O}(n^2)$

Diese Struktur erfüllt alle Dinge, die wir aus dem Interface haben wollten, aber...

- wir benutzen `reverse`, welchen Laufzeit $\mathcal{O}(n)$ hat.
Dies zieht alle Operationen, die dies benutzen auf $\mathcal{O}(n)$ hoch.

Asymptoten:

- `empty` $\mathcal{O}(1)$
- `append` $\mathcal{O}(1)$
- `prepend` $\mathcal{O}(1)$
- `getFirst` $\mathcal{O}(n)$
- `getLast` $\mathcal{O}(n)$
- `iterate` $\mathcal{O}(n^2)$

Die $\mathcal{O}(n)$ -Operation kommt zwar nur selten vor, aber wenn wir z.B. abwechselnd `getFirst` und `getLast` machen, machen wir diese teure Operation jedes mal.

Gibt es da nichts besseres?

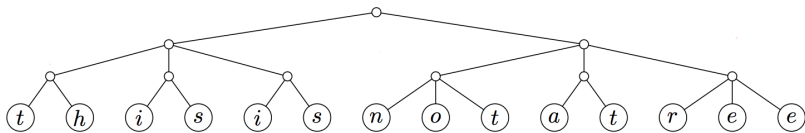
Gibt es da nichts besseres?

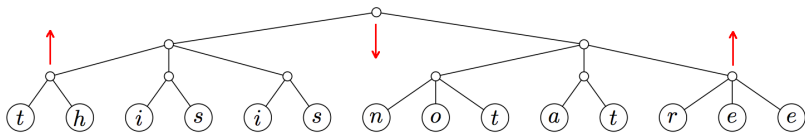
Ja, aber das ist nicht ganz so simpel.

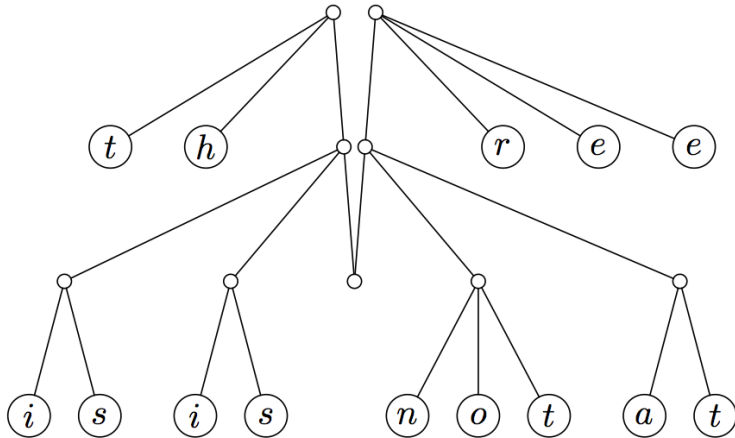
Gibt es da nichts besseres?

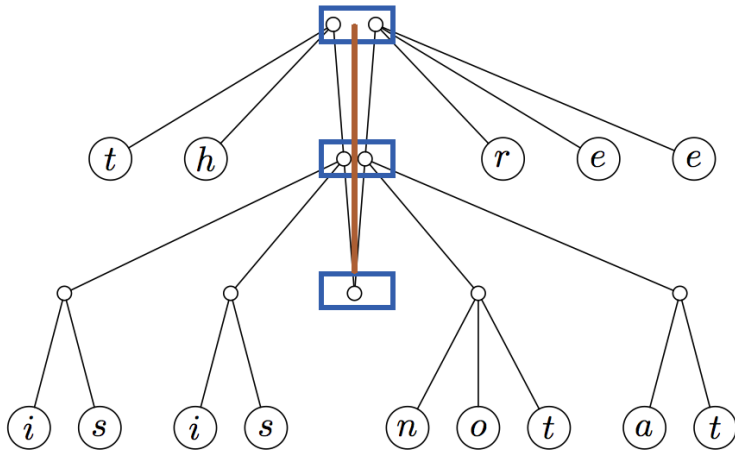
Ja, aber das ist nicht ganz so simpel.

Wir müssen dazu einen Exkurs in Bäume machen. Wir starten mit einem klassischen 2-3-Tree, also einem Baum mit entweder zwei oder drei Kind-Knoten pro Elternknoten.









So ein Baum hat in jeder Ebene ein Präfix, den Rest des Baumes und ein Suffix.

So ein Baum hat in jeder Ebene ein Präfix, den Rest des Baumes und ein Suffix.

In der ersten Ebene hat er zwei oder drei Kind-Knoten

In der zweiten Ebene einen oder zwei Kind-Bäume der Tiefe 1

In der dritten Ebene einen oder zwei Kind-Bäume der Tiefe 2 etc.

So ein Baum hat in jeder Ebene ein Präfix, den Rest des Baumes und ein Suffix.

In der ersten Ebene hat er zwei oder drei Kind-Knoten

In der zweiten Ebene einen oder zwei Kind-Bäume der Tiefe 1

In der dritten Ebene einen oder zwei Kind-Bäume der Tiefe 2 etc.

Wir nennen Präfixe und Suffixe nun *Affixe* und sagen, dass diese bis zu vier Elemente haben können. Dies bringt uns später Vorteile bei der Laufzeit.

Kurz in Code, was wir bisher definiert haben:

```
data Node a = Branch2 a a  
            | Branch3 a a a  
deriving Show
```

Kurz in Code, was wir bisher definiert haben:

```
data Node a = Branch2 a a
            | Branch3 a a a
            deriving Show

data Affix a = One a
            | Two a a
            | Three a a a
            | Four a a a a
            deriving Show
```

Kurz in Code, was wir bisher definiert haben:

```
data Node a = Branch2 a a
            | Branch3 a a a
            deriving Show
```

```
data Affix a = One a
             | Two a a
             | Three a a a
             | Four a a a a
             deriving Show
```

```
data FingerTree a
= Empty      -- We can have empty trees.
| Single a   -- We need a special case for trees of size one.
-- The common case with a prefix, suffix, and a deeper tree.
| Deep {
  prefix :: Affix a,           -- Values on the left.
  deeper  :: FingerTree (Node a), -- The deeper finger tree,
                                           -- storing deeper 2-3 trees.
  suffix  :: Affix a           -- Values on the right.
}
deriving Show
```

Was haben wir hiermit?

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene Deep fügt eine weitere Node hinzu

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene Deep fügt eine weitere Node hinzu

FingerTree (Node a) hat einen Affix von $2 \cdot 1$ (One (Branch2 a)) bis $3 \cdot 4$ ((Four (Branch3 a)) Elementen

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene Deep fügt eine weitere Node hinzu

FingerTree (Node a) hat einen Affix von $2 \cdot 1$ (One (Branch2 a)) bis $3 \cdot 4$ ((Four (Branch3 a)) Elementen

FingerTree (Node (Node a)) hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene Deep fügt eine weitere Node hinzu

FingerTree (Node a) hat einen Affix von $2 \cdot 1$ (One (Branch2 a)) bis $3 \cdot 4$ ((Four (Branch3 a)) Elementen

FingerTree (Node (Node a)) hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen

etc.

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene Deep fügt eine weitere Node hinzu
FingerTree (Node a) hat einen Affix von $2 \cdot 1$ (One (Branch2 a)) bis $3 \cdot 4$ ((Four (Branch3 a)) Elementen
FingerTree (Node (Node a)) hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen
etc.
- Wir haben auf jeder Ebene bis zu 3x mehr Elemente als in den vorhergehenden.

Was haben wir hiermit?

- Einen Baum, der uns die „Enden“ direkt präsentiert
- Jede Ebene Deep fügt eine weitere Node hinzu
FingerTree (Node a) hat einen Affix von $2 \cdot 1$ (One (Branch2 a)) bis $3 \cdot 4$ ((Four (Branch3 a)) Elementen
FingerTree (Node (Node a)) hat einen Affix von $2 \cdot 2 \cdot 1$ bis $3 \cdot 3 \cdot 4$ Elementen
etc.
- Wir haben auf jeder Ebene bis zu 3x mehr Elemente als in den vorhergehenden.
- Der Baum ist automatisch balanciert (nicht perfekt), da jede Ebene sowohl Präfix als auch Suffix-Elemente haben muss

Wie fügen wir nun Elemente ein?

Wie fügen wir nun Elemente ein?

```
infixr 5 <|  
(<|) :: a -> FingerTree a -> FingerTree a  
  
x <| Empty = Single x  
x <| Single y = Deep (One x) Empty (One y)  
x <| Deep (Four a b c d) deeper suffix =  
  Deep (Two x a) (node <| deeper) suffix  
  where  
    node = Branch3 b c d  
x <| tree = tree { prefix = affixPrepend x $ prefix tree }
```

geben eine Funktion `affixPrepend`, die aus einem `One` ein `Two` macht etc.

Analog hinten

```
infixr 5 |>
```

```
(|>) :: FingerTree a -> a -> FingerTree a
```

```
Empty |> x = Single x
```

```
Single y |> x = Deep (One y) Empty (One x)
```

```
Deep prefix deeper (Four a b c d) |> x =  
  Deep prefix (deeper |> node) (Two d x)
```

```
  where
```

```
    node = Branch3 a b c
```

```
tree |> x = tree { suffix = affixAppend x $ suffix tree }
```

gegeben eine Funktion `affixAppend`, die aus einem `One` ein `Two` macht etc.

Aber wie schaut denn nun die Laufzeit aus? Dieses Einfügen führt doch im schlimmsten Falle dazu, dass alle Ebenen angefasst und neu aufgebaut werden müssen!

Aber wie schaut denn nun die Laufzeit aus? Dieses Einfügen führt doch im schlimmsten Falle dazu, dass alle Ebenen angefasst und neu aufgebaut werden müssen!
Richtig. Aber wie häufig?

Aber wie schaut denn nun die Laufzeit aus? Dieses Einfügen führt doch im schlimmsten Falle dazu, dass alle Ebenen angefasst und neu aufgebaut werden müssen!

Richtig. Aber wie häufig?

Jede Operation macht eine $\mathcal{O}(1)$ -Operation in der obersten Ebene. Mit Wahrscheinlichkeit $\frac{1}{2}$ machen wir auch in Ebene 2 eine $\mathcal{O}(1)$ -Operation.

Generell machen wir für m Einfüge-Operationen (von derselben Seite)

$$T = m + \frac{1}{2}m + \frac{1}{4}m + \dots = \sum_{i=0}^m \frac{1}{2^i}m$$

Generell machen wir für m Einfüge-Operationen (von derselben Seite)

$$T = m + \frac{1}{2}m + \frac{1}{4}m + \dots = \sum_{i=0}^m \frac{1}{2^i}m$$

Operationen, was uns allen bekannt vorkommen sollte. Für $m \rightarrow \infty$ Einfügeoperationen brauchen wir folglich

$$m \cdot \lim_{m \rightarrow \infty} \sum_{i=0}^m \frac{1}{2^i} = m \cdot 2$$

Operationen, welches uns amortisiert pro Operation $\mathcal{O}(1)$ kostet.

Das letzte bzw. erste Element bekommen ist ähnlich kompliziert und mit derselben Argumentation kommt man auch hier auf eine Laufzeit von $\mathcal{O}(1)$ für beide Operationen.

Das letzte bzw. erste Element bekommen ist ähnlich kompliziert und mit derselben Argumentation kommt man auch hier auf eine Laufzeit von $\mathcal{O}(1)$ für beide Operationen.

Wir werden hier nur kurz den Code angucken, damit er auf den Folien ist. Weitere Informationen findet man in dem (sehr guten!) Blogpost von Andrew Gibiansky unter <http://andrew.gibiansky.com/blog/haskell/finger-trees/>

Zunächst definieren wir uns eine View-Datenstruktur. Diese ist theoretisch isomorph zu Maybe, macht aber im Kontext der Verwendung Sinn.

```
data View a = Nil | View a (FingerTree a)  
  deriving Show
```

```
viewl :: FingerTree a -> View a
viewl Empty = Nil
viewl (Single x) = View x Empty
viewl (Deep (One x) deeper suffix) = View x rest
  where
    rest =
      case viewl deeper of
        View node rest' ->
          Deep (fromList $ toList node) rest' suffix
        Nil -> case suffix of
          (One x) -> Single x
          (Two x y) -> Deep (One x) Empty (One y)
          (Three x y z) -> Deep (Two x y) Empty (One z)
          (Four x y z w) -> Deep (Three x y z) Empty (One w)
viewl (Deep prefix deeper suffix) =
  View first $ Deep (fromList rest) deeper suffix
  where
    first:rest = toList prefix
```

mit fromList und toList definiert auf Affix

Der Code für `viewr` ist analog hierzu.

Der Code für `viewr` ist analog hierzu.

Wir haben noch das Verketten von zwei Finger-Trees:

Da wir in $\mathcal{O}(1)$ `view` und `append` machen können, können wir in $\mathcal{O}(m)$ Zeit einen Baum der Länge m an einen Baum der Länge n anhängen.

Der Code für `viewr` ist analog hierzu.

Wir haben noch das Verketten von zwei Finger-Trees:

Da wir in $\mathcal{O}(1)$ `view` und `append` machen können, können wir in $\mathcal{O}(m)$ Zeit einen Baum der Länge m an einen Baum der Länge n anhängen.

Dies geht aber auch geschickter über die Struktur des Trees. Für den Code sei auf die Literatur verwiesen. Hier nur die Laufzeit: $\mathcal{O}(\log(\min(n, m)))$.

Mit dieser Struktur geht sogar noch (viel!) mehr.

Mit dieser Struktur geht sogar noch (viel!) mehr.

Gegeben einen Monoid m können wir Annotationen von Typ m hinzufügen.

Mit dieser Struktur geht sogar noch (viel!) mehr.

Gegeben einen Monoid m können wir Annotationen von Typ m hinzufügen.

Wenn wir dann weiterhin so etwas wie

```
class Monoid v => Measured a v where  
  measure :: a -> v
```

vorraussetzen, dann ergibt sich insgesamt

```
data Node v a = Branch3 v a a a
              | Branch2 v a a
              deriving Show
```

```
data FingerTree v a
  = Empty
  | Single a
  | Deep {
    annotation :: v, -- Add an annotation to each branch.
    prefix    :: Affix a,
    deeper    :: FingerTree v (Node v a),
    suffix    :: Affix a
  }
  deriving Show
```

für die Definition des FingerTrees.

```
data Node v a = Branch3 v a a a
              | Branch2 v a a
              deriving Show
```

```
data FingerTree v a
= Empty
| Single a
| Deep {
  annotation :: v, -- Add an annotation to each branch.
  prefix    :: Affix a,
  deeper    :: FingerTree v (Node v a),
  suffix    :: Affix a
}
deriving Show
```

für die Definition des FingerTrees.

Alle bisherigen Operationen müssen natürlich angepasst werden, um die Annotationen mit durchzuschleifen. An der Laufzeit ändert sich nichts.

Einen FingerTree kann man dann messen durch

```
instance Measured a v => Measured (FingerTree v a) v where
  measure Empty = empty
  measure (Single x) = measure x
  measure tree = annotation tree
```

```
instance Measured a v => Measured (Node v a) v where
  measure (Branch2 v _ _) = v
  measure (Branch3 v _ _ _) = v
```

Frage: Was bringt uns das Ganze?

Frage: Was bringt uns das Ganze?
Mittels

```
-- Monoidal size - all leaves have Size 1.  
newtype Size = Size Int deriving (Show, Eq, Ord)
```

```
-- Storage for our values.  
newtype Value a = Value a deriving Show
```

```
-- Sizes just add normally.  
instance Monoid Size where  
  mempty = Size 0  
  Size x <> Size y = Size $ x + y
```

```
-- All values just have size one.  
instance Measured (Value a) Size where  
  measure _ = Size 1
```

Nummerieren wir die Elemente durch. Somit können wir in $\mathcal{O}(\log(n))$ auf das n -te Element zugreifen!

Frage: Was bringt uns das Ganze (2)?

Frage: Was bringt uns das Ganze (2)?
Mittels

```
data Prioritized a = Prioritized {  
    priority :: Int,  
    item    :: a }  
  
data Priority = NegativeInfinity | Priority Int deriving Eq  
instance Monoid Priority where  
    NegativeInfinity <> x = x  
    x <> NegativeInfinity = x  
    (Priority x) <> (Priority y) = Priority $ max x y  
    empty = NegativeInfinity  
instance Measured (Prioritized a) Priority where  
    measure = Priority . priority  
newtype PriorityQueue a = PriorityQueue (FingerTree Priority  
                                         (Prioritized a))
```

wandeln wir den FingerTree in eine Priority-Queue, wo wir schnell auf das Element mit der höchsten Priorität zugreifen können.

Frage: Was bringt uns das Ganze (3)?

Frage: Was bringt uns das Ganze (3)?

Da wir Objekte schnell lokalisieren können (in $\mathcal{O}(\log(n))$), können wir auch einen schnellen Split an so einer Stelle machen.

Was haben wir gelernt?

Was haben wir gelernt?

Die simpelste Lösung ist nicht immer die Beste, aber wir haben ausgehend von doppelt verketteten Listen eine gute Datenstruktur gefunden.

Was haben wir gelernt?

Die simpelste Lösung ist nicht immer die Beste, aber wir haben ausgehend von doppelt verketteten Listen eine gute Datenstruktur gefunden.

Abschließend ein kurzer Überblick über Laufzeiten gängiger Strukturen.

Operation	Amortized Bounds			
	Finger Tree	Annotated 2-3 Tree	List	Vector
cons/snoc	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)/\mathcal{O}(n)$	$\mathcal{O}(n)$
viewl/viewr	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)/\mathcal{O}(n)$	$\mathcal{O}(1)$
measure/lenth	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
append	$\mathcal{O}(\log \min(l_1, l_2))$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n + m)$
split	$\mathcal{O}(\log \min(n, l - n))$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
fromList/toList/reverse	$\mathcal{O}(l)/\mathcal{O}(l)/\mathcal{O}(l)$	$\mathcal{O}(l)$	$\mathcal{O}(1)/\mathcal{O}(1)/\mathcal{O}(n)$	$\mathcal{O}(n)$
index	$\mathcal{O}(\log \min(n, l - n))$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

FingerTrees sind die Datenstruktur, wenn man viele Anfüge und Entfernen-Operationen hat. Daher basiert auch alles in `Data.Sequence` intern auf FingerTrees.

FingerTrees sind die Datenstruktur, wenn man viele Anfüge und Entfernen-Operationen hat. Daher basiert auch alles in `Data.Sequence` intern auf FingerTrees.

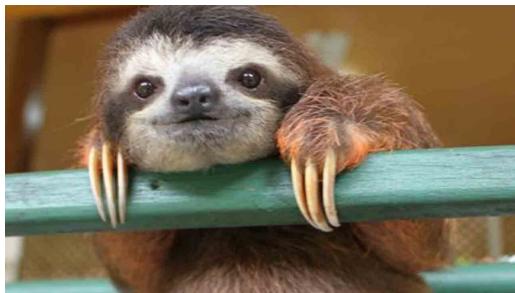
Auch sind sie Doppelt-Verketteten Listen in allen praktischen Anwendungsbereichen überlegen, da sie auch effizienten Random-Access erlauben ($\mathcal{O}(\log n)$ statt $\mathcal{O}(n)$) und sich in $\mathcal{O}(\log n)$ zerteilen lassen (statt $\mathcal{O}(n)$).

FingerTrees sind die Datenstruktur, wenn man viele Anfüge und Entfernen-Operationen hat. Daher basiert auch alles in `Data.Sequence` intern auf FingerTrees.

Auch sind sie Doppelt-Verketteten Listen in allen praktischen Anwendungsbereichen überlegen, da sie auch effizienten Random-Access erlauben ($\mathcal{O}(\log n)$ statt $\mathcal{O}(n)$) und sich in $\mathcal{O}(\log n)$ zerteilen lassen (statt $\mathcal{O}(n)$).

Es gibt noch weitergehende Strukturen, die hierauf aufbauen. Edward Kmett hat z.B. Ropes vorgestellt, welche um einen skalaren Faktor besser und dabei Speicher- und Cachefreundlicher (0.7-30x weniger Speicheraufwand, weniger GC-Zeit) sind und unboxed Strukturen erlauben.

Laziness



Was macht Laziness so besonders?

Was macht Laziness so besonders?

- Trennung von Definition und Evaluation

Was macht Laziness so besonders?

- Trennung von Definition und Evaluation
Wir können Dinge separat definieren, aber gemeinsam berechnen

Was macht Laziness so besonders?

- Trennung von Definition und Evaluation
Wir können Dinge separat definieren, aber gemeinsam berechnen
- Vermeidung von unnötigen Berechnungen

Was macht Laziness so besonders?

- Trennung von Definition und Evaluation
Wir können Dinge separat definieren, aber gemeinsam berechnen
- Vermeidung von unnötigen Berechnungen

```
select :: Ord o => Int -> [o] -> [o]  
select n = take n . sort
```

Was macht Laziness so besonders?

- Trennung von Definition und Evaluation
Wir können Dinge separat definieren, aber gemeinsam berechnen

- Vermeidung von unnötigen Berechnungen

```
select :: Ord o => Int -> [o] -> [o]  
select n = take n . sort
```

hört mit dem sortieren auf, sobald die ersten n kleinsten Elemente gefunden wurde.

Was macht Laziness so besonders?

- Trennung von Definition und Evaluation
Wir können Dinge separat definieren, aber gemeinsam berechnen

- Vermeidung von unnötigen Berechnungen

```
select :: Ord o => Int -> [o] -> [o]
select n = take n . sort
```

hört mit dem sortieren auf, sobald die ersten n kleinsten Elemente gefunden wurde.

Wie das hinzufügen eines `break` in imperativem Code (innerhalb der Sortierfunktion)

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben
 - auf sie Pattern-Matchen

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben
 - auf sie Pattern-Matchen
 - sie zu größeren Dingen komponieren

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben
 - auf sie Pattern-Matchen
 - sie zu größeren Dingen komponieren
- Lazy Datenstrukturen brauchen nicht komplett im Speicher liegen, sondern werden erstellt, wenn sie benötigt werden

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben
 - auf sie Pattern-Matchen
 - sie zu größeren Dingen komponieren
- Lazy Datenstrukturen brauchen nicht komplett im Speicher liegen, sondern werden erstellt, wenn sie benötigt werden
 - werden sofort GC, wenn benutzt

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben
 - auf sie Pattern-Matchen
 - sie zu größeren Dingen komponieren
- Lazy Datenstrukturen brauchen nicht komplett im Speicher liegen, sondern werden erstellt, wenn sie benötigt werden
 - werden sofort GC, wenn benutzt
 - `factorial n = product [1..n]`
wird zu einem for-loop mit constant-memory usage

- Lazy Datenstrukturen sind im Prinzip eine Steuerung des Kontrollflusses.
z.B. entspricht eine Liste einem For-Loop in imperativen Programmiersprachen
- Lazy Datenstrukturen sind aber First-Class-Citizens in Haskell, sprich
 - wir können sie an Funktionen übergeben
 - auf sie Pattern-Matchen
 - sie zu größeren Dingen komponieren
- Lazy Datenstrukturen brauchen nicht komplett im Speicher liegen, sondern werden erstellt, wenn sie benötigt werden
 - werden sofort GC, wenn benutzt
 - `factorial n = product [1..n]`
wird zu einem for-loop mit constant-memory usage
 - `fold . f . unfold`
ist klassisch in der Datenverarbeitung

Wenn eine Liste ein Loop ist, dann sind verschachtelte Listen verschachtelte Loops. Dies ermöglicht uns non-deterministische Programmierung.

Wenn eine Liste ein Loop ist, dann sind verschachtelte Listen verschachtelte Loops. Dies ermöglicht uns non-deterministische Programmierung.

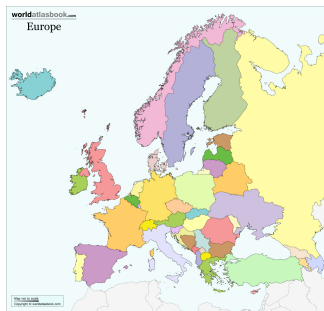
Wir kennen das schon von der Monaden-Instanz von List:

```
do a <- [1..10]
  b <- [1..10]
  guard (a /= b && a + b = 7)
  return (a,b)
```

Es existieren nie alle 100 Kombinationen von a und b im RAM, sondern nur diejenigen, die das Prädikat überleben und die nächste, die probiert wird.

Kartenfärbungsproblem:

Gegeben eine Karte mit Staaten, wollen wir nun eine Einfärbung der Karte errechnen - wobei keine benachbarten Staaten dieselbe Farbe bekommen sollen.



Wenn wir die Funktion

```
step :: Map -> Country -> [Map]
```

haben, die uns einen beliebigen Staat entsprechend den Kriterien einfärbt, dann ist

Wenn wir die Funktion

```
step :: Map -> Country -> [Map]
```

haben, die uns einen beliebigen Staat entsprechend den Kriterien einfärbt, dann ist

```
solutions :: [Map]  
solutions = foldM step blank states
```

die Liste, die alle validen Einfärbungen enthält.

Mittels einem Ausdruck wie
`first = head solutions`
bekommen wir irgendeine Lösung.

Mittels einem Ausdruck wie

```
first = head solutions
```

bekommen wir irgendeine Lösung.

Aber wir können auch Anforderungen an die Lösung haben:

```
some = find gerBlue solutions
```

```
all = filter getBlue solutions
```

Mittels einem Ausdruck wie

```
first = head solutions
```

bekommen wir irgendeine Lösung.

Aber wir können auch Anforderungen an die Lösung haben:

```
some = find gerBlue solutions
```

```
all = filter getBlue solutions
```

Dies gibt uns eine (oder alle) Karten, bei denen Deutschland blau gefärbt ist.

Mittels einem Ausdruck wie

```
first = head solutions
```

bekommen wir irgendeine Lösung.

Aber wir können auch Anforderungen an die Lösung haben:

```
some = find gerBlue solutions
```

```
all = filter getBlue solutions
```

Dies gibt uns eine (oder alle) Karten, bei denen Deutschland blau gefärbt ist.

Durch Laziness haben wir nie alle Lösungen im Speicher sondern immer nur die minimale Anzahl.

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können.

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können. Es gibt ein Paket (<https://github.com/mvr/cf>), welches mit sogenannten continued fractions rechnet.

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können. Es gibt ein Paket (<https://github.com/mvr/cf>), welches mit sogenannten continued fractions rechnet. Eine genaue Erklärung gibt es unter <https://www.youtube.com/watch?v=LJQgYBQFtSE>. Kurz zusammengefasst:

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können. Es gibt ein Paket (<https://github.com/mvr/cf>), welches mit sogenannten continued fractions rechnet. Eine genaue Erklärung gibt es unter <https://www.youtube.com/watch?v=LJQgYBQFtSE>. Kurz zusammengefasst:

- Jede rationale Zahl lässt sich als endliche Liste darstellen

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können. Es gibt ein Paket (<https://github.com/mvr/cf>), welches mit sogenannten continued fractions rechnet. Eine genaue Erklärung gibt es unter <https://www.youtube.com/watch?v=LJQgYBQFtSE>. Kurz zusammengefasst:

- Jede rationale Zahl lässt sich als endliche Liste darstellen
- π , e lassen sich durch unendlich lange Listen darstellen (lazy)

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können. Es gibt ein Paket (<https://github.com/mvr/cf>), welches mit sogenannten continued fractions rechnet. Eine genaue Erklärung gibt es unter <https://www.youtube.com/watch?v=LJQgYBQFtSE>. Kurz zusammengefasst:

- Jede rationale Zahl lässt sich als endliche Liste darstellen
- π , e lassen sich durch unendlich lange Listen darstellen (lazy)
- alle Operationen $+$ $-$ \times $/$ konsumieren soviel von der Liste, wie sie brauchen

Laziness kann auch Präzision in Datenstrukturen nachliefern, weil wir eine unendlich genaue Datenstruktur definieren können. Es gibt ein Paket (<https://github.com/mvr/cf>), welches mit sogenannten continued fractions rechnet. Eine genaue Erklärung gibt es unter <https://www.youtube.com/watch?v=LJQgYBQFtSE>. Kurz zusammengefasst:

- Jede rationale Zahl lässt sich als endliche Liste darstellen
- π , e lassen sich durch unendlich lange Listen darstellen (lazy)
- alle Operationen $+$ $-$ \times $/$ konsumieren soviel von der Liste, wie sie brauchen
- am ENDE legt man die Genauigkeit fest, die man gerne hätte und alle Zahlen bis da hin werden nur so weit berechnet, wie man für die gegebene Genauigkeit braucht.

Laziness ist ein „built-in controlled side-effect“, welcher unter unserer Programmierschwelle liegt. Der Compiler kümmert sich also darum, dass

Laziness ist ein „built-in controlled side-effect“, welcher unter unserer Programmierschwelle liegt. Der Compiler kümmert sich also darum, dass

- Wir lazy Werte höchstens 1x berechnen und ggf. cachen, wenn wir sie später nochmal brauchen
- alles deterministisch läuft (obwohl wir nicht-deterministisch schreiben können)
- alles thread-safe abläuft (und wir somit einfach parallelisieren können)

Laziness ist ein „built-in controlled side-effect“, welcher unter unserer Programmierschwelle liegt. Der Compiler kümmert sich also darum, dass

- Wir lazy Werte höchstens 1x berechnen und ggf. cachen, wenn wir sie später nochmal brauchen
- alles deterministisch läuft (obwohl wir nicht-deterministisch schreiben können)
- alles thread-safe abläuft (und wir somit einfach parallelisieren können)

All das bekommen wir durch Laziness kostenlos über das Runtime-System und wir können hierbei keine Fehler mehr machen (sofern wir uns von unsafe-Functions fernhalten).

Beispiel:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Beispiel:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Wenn wir nun die n -te Fibonnaci-Zahl haben wollen, dann

- errechnen wir die nächste Zahl aus den vorhergehenden

Beispiel:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Wenn wir nun die n -te Fibonnaci-Zahl haben wollen, dann

- errechnen wir die nächste Zahl aus den vorhergehenden
- sammelt der GC die Zahlen wieder ein, die wir nicht brauchen

Beispiel:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Wenn wir nun die n -te Fibonnaci-Zahl haben wollen, dann

- errechnen wir die nächste Zahl aus den vorhergehenden
- sammelt der GC die Zahlen wieder ein, die wir nicht brauchen
- haben wir maximal zwei Integer und die Berechnung für die nächste Zahl im Speicher

Beispiel:

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Wenn wir nun die n -te Fibonnaci-Zahl haben wollen, dann

- errechnen wir die nächste Zahl aus den vorhergehenden
- sammelt der GC die Zahlen wieder ein, die wir nicht brauchen
- haben wir maximal zwei Integer und die Berechnung für die nächste Zahl im Speicher

Außerdem können wir diese (unendliche) Liste auch mit allen anderen Funktionen manipulieren.

Generell nennt man das Zwischenspeichern von Teilergebnissen
Memoization.

Generell nennt man das Zwischenspeichern von Teilergebnissen
Memoization.

Dieses kann entweder automatisch geschehen oder muss manuell gemacht werden.

Generell nennt man das Zwischenspeichern von Teilergebnissen **Memoization**.

Dieses kann entweder automatisch geschehen oder muss manuell gemacht werden.

In Haskell können wir uns nicht immer auf den Compiler verlassen das automatisch zu tun, aber es gibt Pakete, die das garantieren:

- `data-memocombinators`
- `MemoTrie`
- `infinite lazy trees`

Ein weiterer Fall für die Anwendung von Laziness gibt es in der **dynamischen Programmierung**.

Ein weiterer Fall für die Anwendung von Laziness gibt es in der **dynamischen Programmierung**.

Hier wird eine Beschleunigung des Programmes dadurch erreicht, dass man gezielt Zwischenspeicher (klassischerweise Arrays) anlegt und immer wieder auf diese zugreift.

Ein weiterer Fall für die Anwendung von Laziness gibt es in der **dynamischen Programmierung**.

Hier wird eine Beschleunigung des Programmes dadurch erreicht, dass man gezielt Zwischenspeicher (klassischerweise Arrays) anlegt und immer wieder auf diese zugreift.

So kann man mit Hilfe von „Bellmans Principle“ (auf das wir nicht weiter eingehen) einen exponentiellen Suchraum in polynomieller Zeit auf ein Ergebnis hin untersuchen.

Häufig wollen wir in der dynamischen Programmierung also einen Lookup-Array haben.

Häufig wollen wir in der dynamischen Programmierung also einen Lookup-Array haben.

Wir können hier einfach einen Array mit der Berechnung lazy erzeugen und das RTS kümmert sich darum, dass jeder Wert zur richtigen Zeit gefüllt ist und keiner doppelt berechnet wird.

Häufig wollen wir in der dynamischen Programmierung also einen Lookup-Array haben.

Wir können hier einfach einen Array mit der Berechnung lazy erzeugen und das RTS kümmert sich darum, dass jeder Wert zur richtigen Zeit gefüllt ist und keiner doppelt berechnet wird.

Beispiel:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fibs ! (n-1) + fibs ! (n-2)
  where
    fibs = Array (0,n) [fib i | i <- [0..n]]
```

Häufig wollen wir in der dynamischen Programmierung also einen Lookup-Array haben.

Wir können hier einfach einen Array mit der Berechnung lazy erzeugen und das RTS kümmert sich darum, dass jeder Wert zur richtigen Zeit gefüllt ist und keiner doppelt berechnet wird.

Beispiel:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fibs ! (n-1) + fibs ! (n-2)
  where
    fibs = Array (0,n) [fib i | i <- [0..n]]
```

- Für Fibonacci overkill, da wir immer nur zwei Elemente „nach hinten“ schauen müssen

Häufig wollen wir in der dynamischen Programmierung also einen Lookup-Array haben.

Wir können hier einfach einen Array mit der Berechnung lazy erzeugen und das RTS kümmert sich darum, dass jeder Wert zur richtigen Zeit gefüllt ist und keiner doppelt berechnet wird.

Beispiel:

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fibs ! (n-1) + fibs ! (n-2)
  where
    fibs = Array (0,n) [fib i | i <- [0..n]]
```

- Für Fibonacci overkill, da wir immer nur zwei Elemente „nach hinten“ schauen müssen
- Für Aufgaben, die beliebig „nach hinten“ sehen müssen (Strig-Edit-Distance, etc.) ist dies aber die beste Lösung.

Was haben wir nun von Laziness?

Was haben wir nun von Laziness?

- Modularität

Was haben wir nun von Laziness?

- Modularität

Wir können Definition und Evaluation trennen

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss
Wir können über lazy Datenstrukturen den Kontrollfluss ändern und später definieren, was wir eigentlich haben wollen

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss
Wir können über lazy Datenstrukturen den Kontrollfluss ändern und später definieren, was wir eigentlich haben wollen
- Präzision

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss
Wir können über lazy Datenstrukturen den Kontrollfluss ändern und später definieren, was wir eigentlich haben wollen
- Präzision
Wir können beliebig präzise rechnen (arbitrary precision floating point, zahlreiche 2D-Anwendungen ähnlich zu Vektorgrafiken)

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss
Wir können über lazy Datenstrukturen den Kontrollfluss ändern und später definieren, was wir eigentlich haben wollen
- Präzision
Wir können beliebig präzise rechnen (arbitrary precision floating point, zahlreiche 2D-Anwendungen ähnlich zu Vektorgrafiken)
- Memoization

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss
Wir können über lazy Datenstrukturen den Kontrollfluss ändern und später definieren, was wir eigentlich haben wollen
- Präzision
Wir können beliebig präzise rechnen (arbitrary precision floating point, zahlreiche 2D-Anwendungen ähnlich zu Vektorgrafiken)
- Memoization
Der Compiler optimiert unseren Code, indem er automatisch sachen cached, welche mehrfach verwendet werden

Was haben wir nun von Laziness?

- Modularität
Wir können Definition und Evaluation trennen
- Kontrollfluss
Wir können über lazy Datenstrukturen den Kontrollfluss ändern und später definieren, was wir eigentlich haben wollen
- Präzision
Wir können beliebig präzise rechnen (arbitrary precision floating point, zahlreiche 2D-Anwendungen ähnlich zu Vektorgrafiken)
- Memoization
Der Compiler optimiert unseren Code, indem er automatisch sachen cached, welche mehrfach verwendet werden

Mehr Informationen/Quellen:

<http://begriffs.com/posts/>

2015-06-17-thinking-with-laziness.html