Zipper

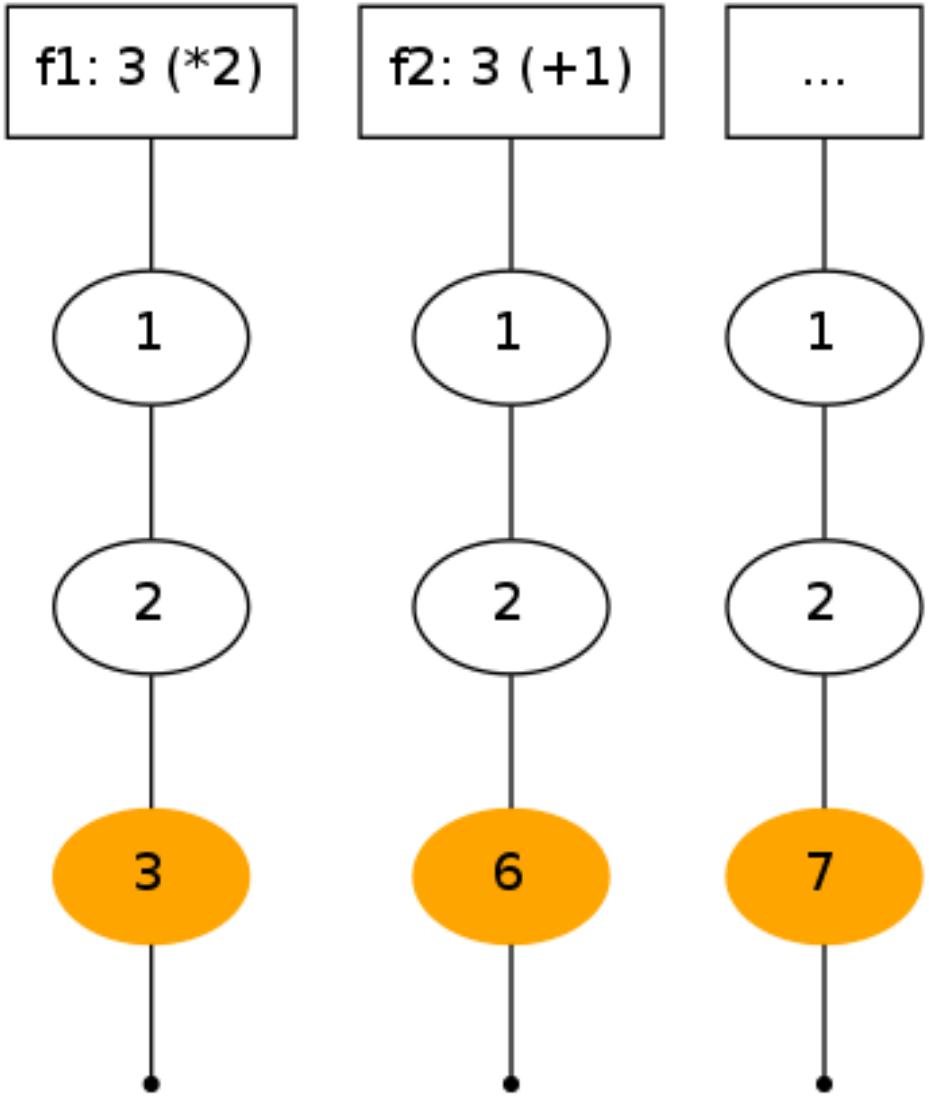
von Julian Wefers und Alexander Mills

Problemstellung

Lange Liste, (mehrfache) Bearbeitung einzelner Elemente

```
data [a] = []
         | a : [a]
```

Problemstellung



Problemstellung

Naive Lösung

```
modify :: Int -> (a -> a) -> [a] -> [a]
```

```
doSomething = modify 3 (+1) . modify 3 (*2)
...
doSomething [1,2,3]
=> [1,2,7]
```

Problemstellung

Naive Lösung

```
modify :: Int -> (a -> a) -> [a] -> [a]
modify 0 f (x:xs) = (f x):xs


doSomething = modify 3 (+1) . modify 3 (*2)
...
doSomething [1,2,3]
=> [1,2,7]
```

Problemstellung

Naive Lösung

```
modify :: Int -> (a -> a) -> [a] -> [a]
modify 0 f (x:xs) = (f x):xs
modify n f (x:xs) = x:(modify (n-1) f xs)

doSomething = modify 3 (+1) . modify 3 (*2)
...
doSomething [1,2,3]
=> [1,2,7]
```

Problemstellung

Effizient?

Problemstellung

Effizient?

```
doSomething2 = modify 10000 (+1)
             . modify 10000 (*2)
             . modify 9999 (+1)
             . modify 9999 (*2)

doSomething2 [1..20000]
```

Zipper für Listen

Datenstruktur, welche:

- Teilliste fokussiert
- Kontext der Teilliste bereitstellt

Zipper für Listen

Datenstruktur, welche:

- Teilliste fokussiert
- Kontext der Teilliste bereitstellt

```
data ListCtx a = Front
               | E (ListCtx a) a
```

Zipper für Listen

Datenstruktur, welche:

- Teilliste fokussiert
- Kontext der Teilliste bereitstellt

```
data ListCtx a = Front
               | E (ListCtx a) a

               ([a], ListCtx a)
```

Zipper für Listen

Datenstruktur, welche:

- Teilliste fokussiert
- Kontext der Teilliste bereitstellt

```
data ListCtx a = Front
               | E (ListCtx a) a

type Zipper a = ([a], ListCtx a)
```

Zipper für Listen

Neue Liste:

```
fromList => front

front :: [a] -> Zipper a
```

Zipper für Listen

Neue Liste:

```
fromList => front

front :: [a] -> Zipper a
front t = (t, Front)

front [1,2,3]
=> ([1,2,3], Front)
```

Zipper für Listen

```
forward :: Zipper a -> Zipper a

back :: Zipper a -> Zipper a

frontmost :: Zipper a -> Zipper a
```

Zipper für Listen

```
forward :: Zipper a -> Zipper a
forward (x:xs, c) =

back :: Zipper a -> Zipper a


frontmost :: Zipper a -> Zipper a
```

Zipper für Listen

```
forward :: Zipper a -> Zipper a
forward (x:xs, c) = (xs, E c x)

back :: Zipper a -> Zipper a
back (xs, E c x) =

frontmost :: Zipper a -> Zipper a
```

Zipper für Listen

```
forward :: Zipper a -> Zipper a
forward (x:xs, c) = (xs, E c x)

back :: Zipper a -> Zipper a
back (xs, E c x) = (x:xs, c)

frontmost :: Zipper a -> Zipper a
frontmost (n, Front) =
frontmost z =
```

Zipper für Listen

```
forward :: Zipper a -> Zipper a
forward (x:xs, c) = (xs, E c x)

back :: Zipper a -> Zipper a
back (xs, E c x) = (x:xs, c)

frontmost :: Zipper a -> Zipper a
frontmost (n, Front) = (n, Front)
frontmost z = frontmost $ back z
```

Zipper für Listen

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (t:ts, c) = ((f t):ts, c)

doSomething = modify (+1) . modify (*2)

doSomething forward $
             forward $ front [1,2,3]
```

Zipper für Listen

```
modifyN :: Int -> (a -> a) ->
           Zipper a -> Zipper a
modifyN 0 f (l, c) = (f l, c)
modifyN n f z
   | n>0 = modifyN (n-1) f (forward z)
   | n<0 = modifyN (n+1) f (back z)

doSomething = modify 0 (+1) . modify 2 (*2)

doSomething front [1,2,3]
```

Zipper für Listen

Effizient?

```
doSomething2 = modifyN 0 (+1)
             . modifyN 1 (*2)
             . modifyN 0 (+1)
             . modifyN 9999 (*2)

doSomething2 $ front [1..20000]
```

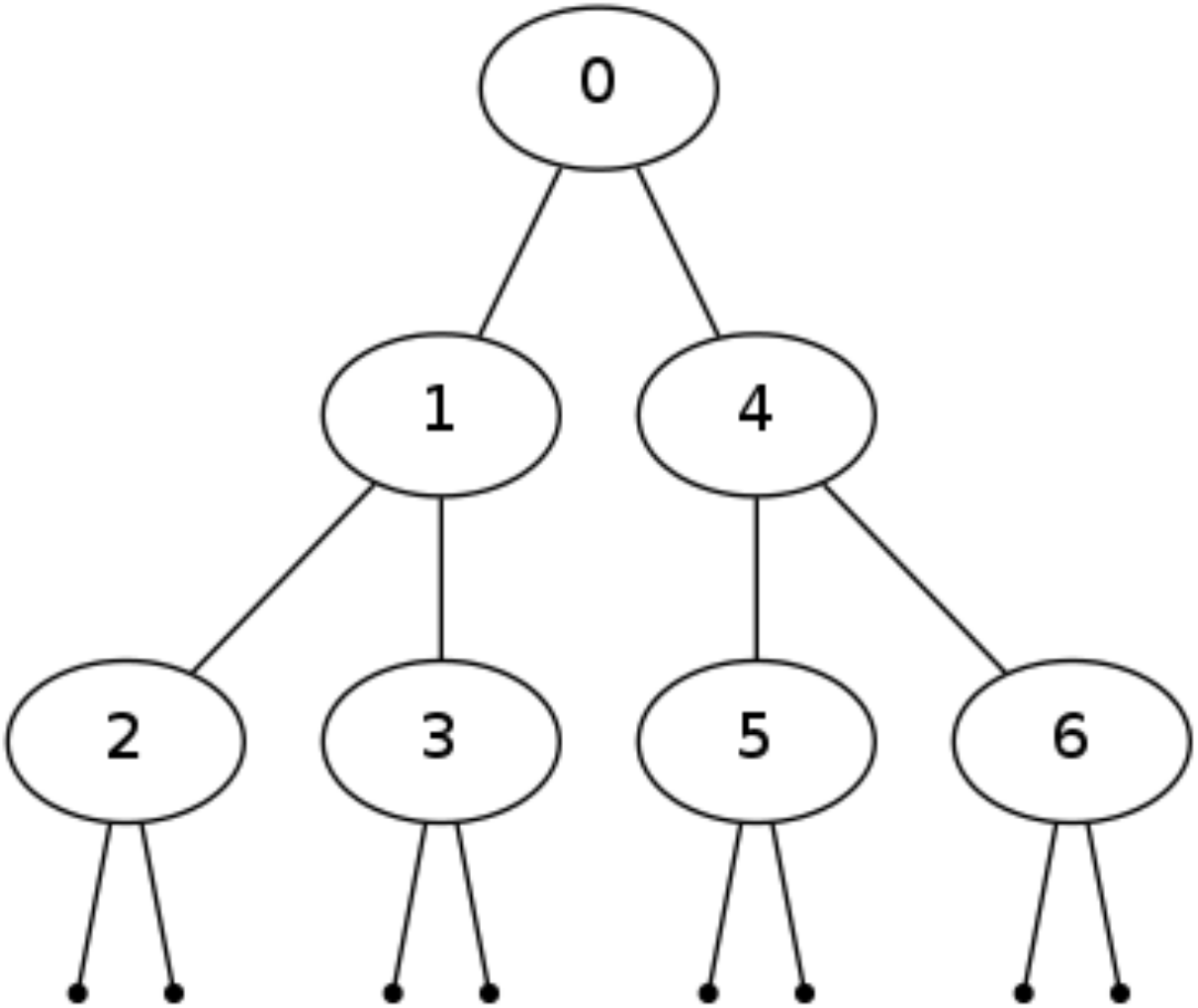
Zipper für Bäume

Zipper für Bäume

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)

exampleTree :: Tree Int
exampleTree = Node 0
              (Node 1
                 (Node 2 Empty Empty)
                 (Node 3 Empty Empty))
              (Node 4
                 (Node 5 Empty Empty)
                 (Node 6 Empty Empty))
```

Zipper für Bäume



Zipper für Bäume

Kontext eines Knotens merken

```
data TreeCtx a = Top
               | ...
               | ...
```

Zipper für Bäume

Kontext eines Knotens merken

```
data TreeCtx a = Top
               | L (TreeCtx a) a (Tree a)
               | R (TreeCtx a) a (Tree a)

type Zipper a = ...
```

Zipper für Bäume

Kontext eines Knotens merken

```
data TreeCtx a = Top
               | L (TreeCtx a) a (Tree a)
               | R (TreeCtx a) a (Tree a)

type Zipper a = (Tree a, TreeCtx a)
```

Zipper für Bäume

```
top :: Tree a -> Zipper a
top t = (t, Top)

left :: Zipper a -> Zipper a


right :: Zipper a -> Zipper a
```

Zipper für Bäume

```
top :: Tree a -> Zipper a
top t = (t, Top)

left :: Zipper a -> Zipper a
left (Node x l r, c) =

right :: Zipper a -> Zipper a
right (Node x l r, c) =
```

Zipper für Bäume

```
top :: Tree a -> Zipper a
top t = (t, Top)

left :: Zipper a -> Zipper a
left (Node x l r, c) = (l, L c x r)

right :: Zipper a -> Zipper a
right (Node x l r, c) = (r, R c x l)
```

Zipper für Bäume

Wiederaufbau des Baumes:

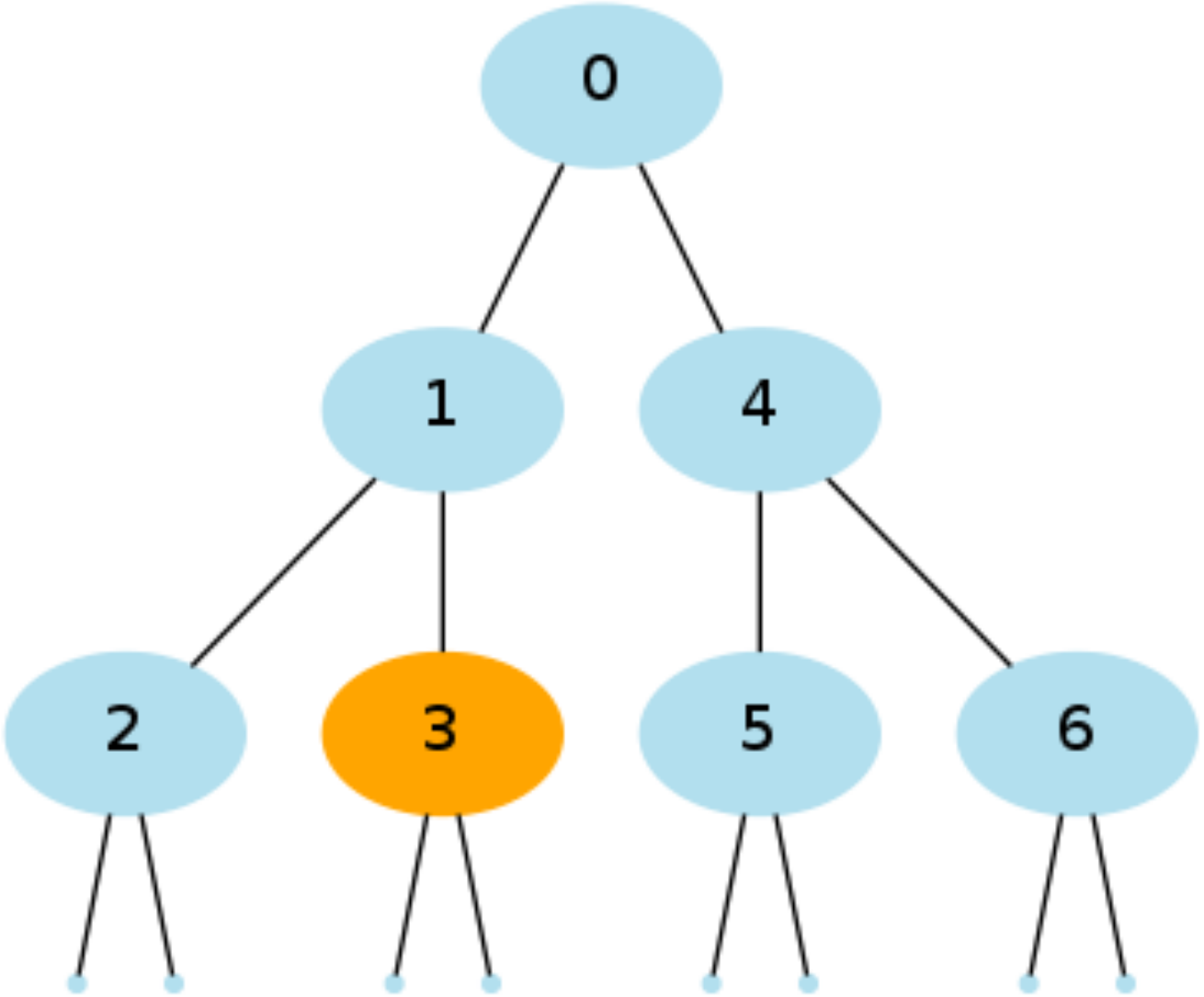
```
up :: (Tree a, TreeCtx a)
up (n, L c x r) = (Node x n r, c)
up (n, R c x l) = (Node x l n, c)

upMost :: Zipper a -> Zipper a
upMost (t, Top) = (t, Top)
upMost x = upMost (up x)
```

Zipper für Bäume

```
right . left . top $ exampleTree
=>
Node 3 Empty Empty,
R (L Top 0 (Node 4
                (Node 5 Empty Empty)
                (Node 6 Empty Empty)))
                1 (Node 2 Empty Empty)
```

Zipper für Bäume



Absicherung

Keine Fehlerbehandlung

```
left (Empty, ...) => ?
up (..., Top) => ?
```

Schlecht

Absicherung

```
left :: Zipper a -> Maybe (Zipper a)
left (Empty, _) = Nothing
left (Node x l r, c) = Just (l, L c x r)

right :: Zipper a -> Maybe (Zipper a)
right (Empty, _) = Nothing
right (Node x l r, c) = Just (r, R c x l)
```

Absicherung

```
up :: Zipper a -> Maybe (Zipper a)
up (_, Top) = Nothing
up (n, L c x r) = Just (Node x n r, c)
up (n, R c x l) = Just (Node x l n, c)
```

Anwendungsbeispiel

Anwendungsbeispiel

```
type Name = String
type Data = String
```

Anwendungsbeispiel

```
type Name = String
type Data = String

data FSItem = File Name Data
            | Folder Name [FSItem]
              deriving Show
```

Anwendungsbeispiel

```
myDisk =
   Folder "root"
      [ File "hello.txt" "Hallo Welt"
      , File "fp.hs" "insert mega cool
                      Haskell Code here"
      , Folder "pics"
           [ File "cat.jpg" "Awwww"
           , File "cat.gif" "OMG so cute!!"
           , File "cat.bmp" "AWWWWWWWWWWWWWW"
           ]
   ]
```

Anwendungsbeispiel

```
data FSCtx = ...
           | ...
             ...
             deriving Show

type FSZipper = ...
```

Anwendungsbeispiel

```
data FSCtx = Root
           | Dir Name [FSItem]
             [FSItem] FSCtx
             deriving Show

type FSZipper = (FSItem, FSCtx)
```

Anwendungsbeispiel

```
top :: FSItem -> FSZipper


up :: FSZipper -> FSZipper
```

Anwendungsbeispiel

```
top :: FSItem -> FSZipper
top x = (x, Root)

up :: FSZipper -> FSZipper
```

Anwendungsbeispiel

```
top :: FSItem -> FSZipper
top x = (x, Root)

up :: FSZipper -> FSZipper
up (x, Dir n l r c) =
   (Folder n (l ++ x:r), c)
```

Anwendungsbeispiel

```
top :: FSItem -> FSZipper
top x = (x, Root)

up :: FSZipper -> FSZipper
up (x, Dir n l r c) =
   (Folder n (l ++ x:r), c)
up z = z
```

Anwendungsbeispiel

```
down :: Name -> FSZipper -> FSZipper
```

Anwendungsbeispiel

```
down :: Name -> FSZipper -> FSZipper
```

```
isName :: Name -> FSItem -> Bool
isName n (Folder fn xs) = n == fn
isName n (File fn d) = n == fn
```

Anwendungsbeispiel

```
down :: Name -> FSZipper -> FSZipper
down n (Folder fn fl, c) =
   (x, Dir fn ls rs c)
   where (ls, x:rs) = break (isName n) fl

isName :: Name -> FSItem -> Bool
isName n (Folder fn xs) = n == fn
isName n (File fn d) = n == fn
```

Anwendungsbeispiel

LIVE Vorführung

Das Ende

Fragen?

Das Ende

Ende