Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Outline I

Übersicht für Heute:

1. Lens
   - Grundidee
   - Motivation & Anwendungen
   - Fortgeschrittenes
2. QuickCheck

Lenses

Was sind „Lenses“und wozu braucht man die?

foo bar

Was sind „Lenses“und wozu braucht man die?

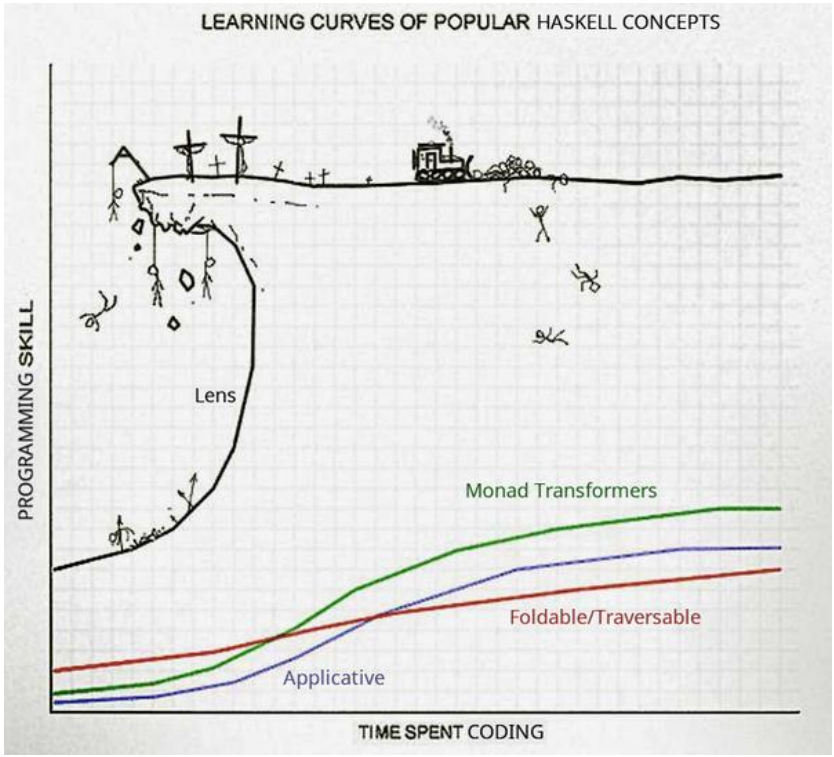
foo bar

Und warum sollte ich jetzt zuhören, wenn eh nur eine Bibliothek vorgestellt wird?

bar baz

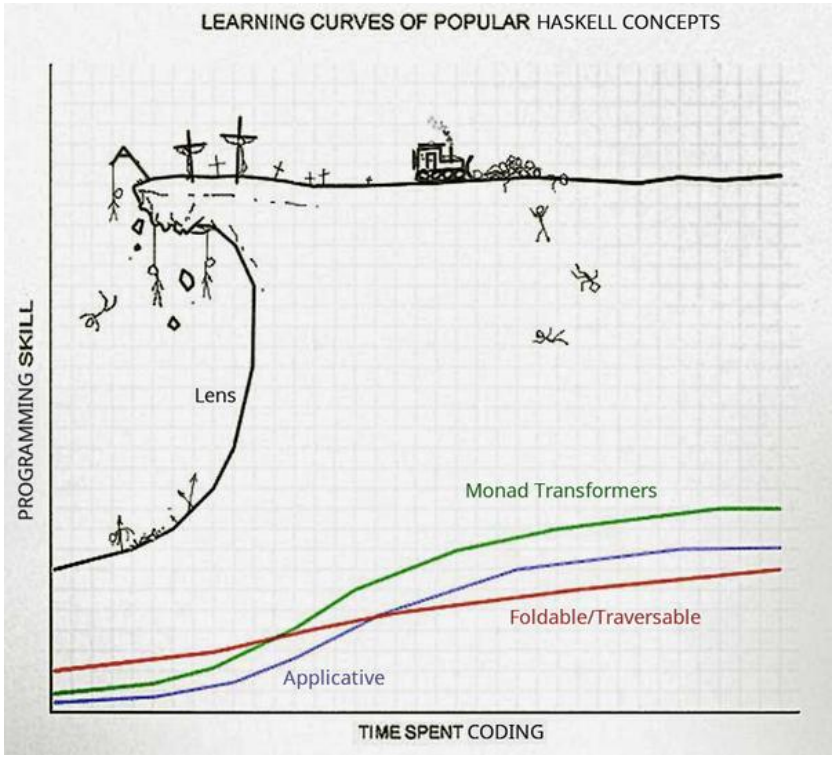
Regel 1: Keine Panik!

Regel 1: Keine Panik!



Sich über die Komplexität der `Lens`-Bibliothek lustig zu machen, ist zu einem gewissen *inside joke* der Community geworden...

Regel 1: Keine Panik!



Sich über die Komplexität der `Lens`-Bibliothek lustig zu machen, ist zu einem gewissen *inside joke* der Community geworden...

Das bedeutet aber auch, dass es (größtenteils) nicht so schlimm ist, wie Leute behaupten.

Die Grundidee:

Eine `Lens` gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

Die Grundidee:

Eine `Lens` gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

Zugriff bedeutet hier...

- lesen, schreiben, modifizieren...

Die Grundidee:

Eine `Lens` gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

Zugriff bedeutet hier...

- lesen, schreiben, modifizieren...
- aber auch falten, traversieren usw.

Die Grundidee:

Eine `Lens` gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

Zugriff bedeutet hier...

- lesen, schreiben, modifizieren...
- aber auch falten, traversieren usw.

`Lens`es sind „first-class values“ (können also umhergereicht, in Datenstrukturen gepackt oder zurückgegeben werden...). Die simple Variante hat den Typ `Lens' s a`.

Die Grundidee:

Eine `Lens` gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

Zugriff bedeutet hier...

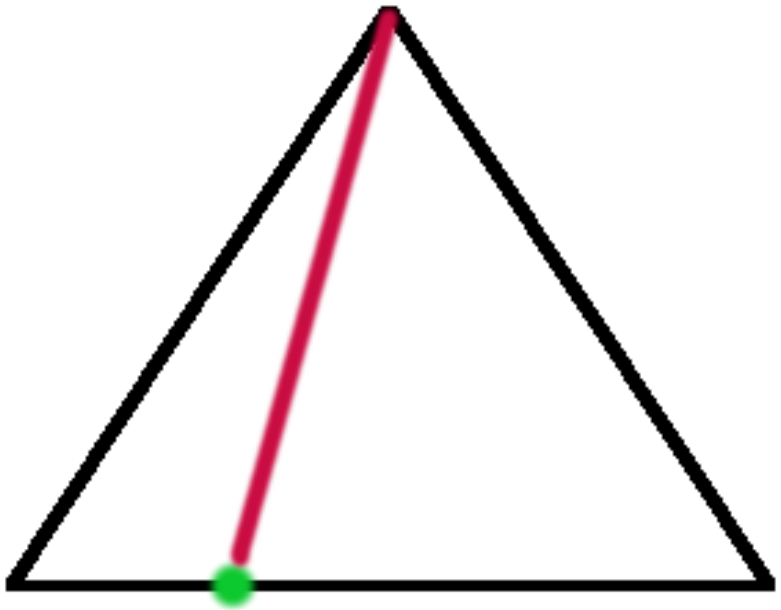
- lesen, schreiben, modifizieren...
- aber auch falten, traversieren usw.

`Lenses` sind „first-class values“ (können also umhergereicht, in Datenstrukturen gepackt oder zurückgegeben werden...). Die simple Variante hat den Typ `Lens' s a`.

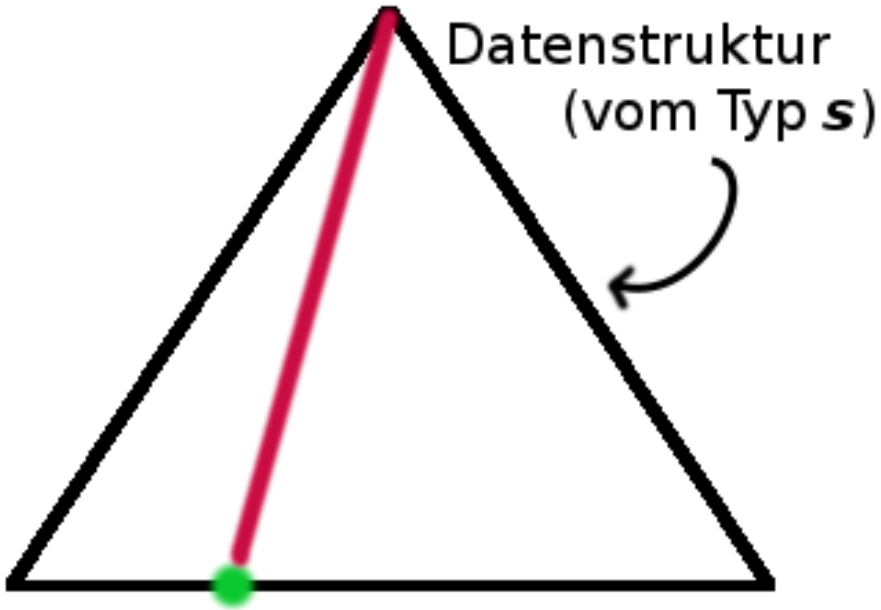
Beispiele:

```
Lens' DateTime Hour
Lens' DateTime Minute
...
```

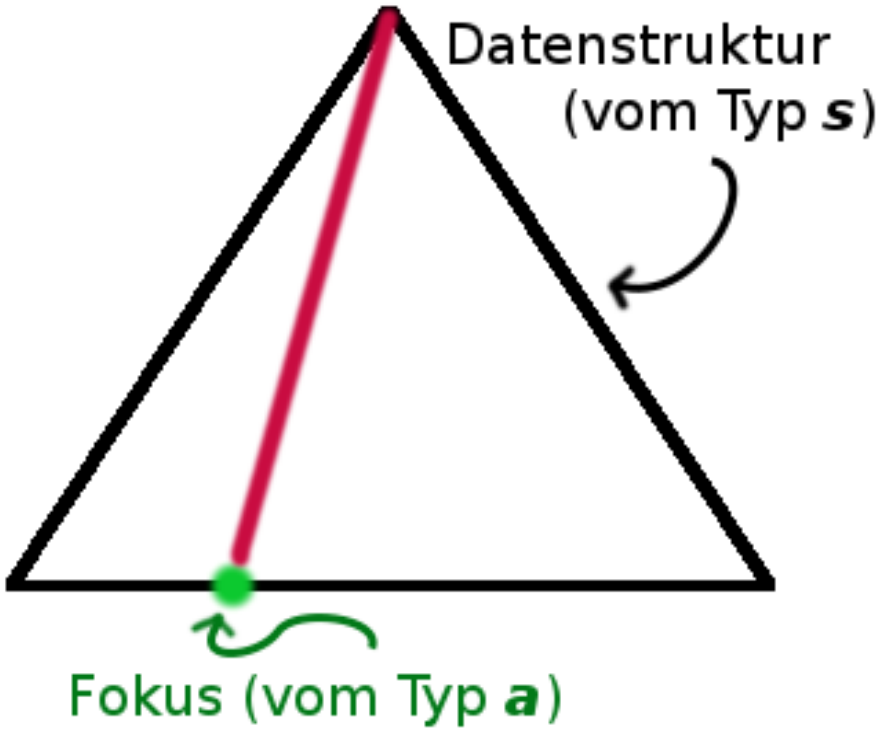
Die Grundidee:



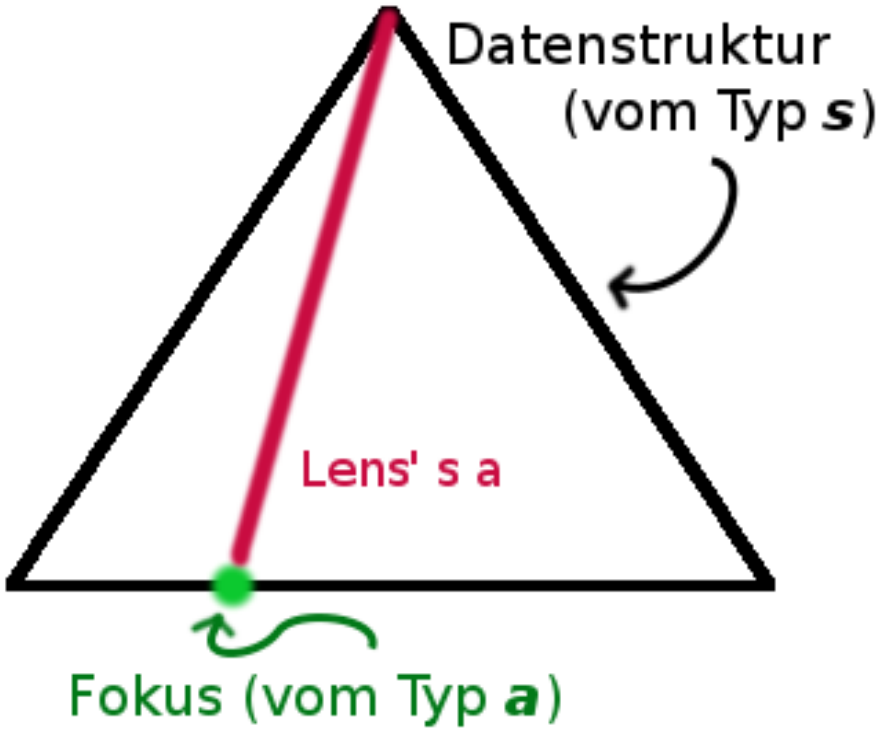
Die Grundidee:

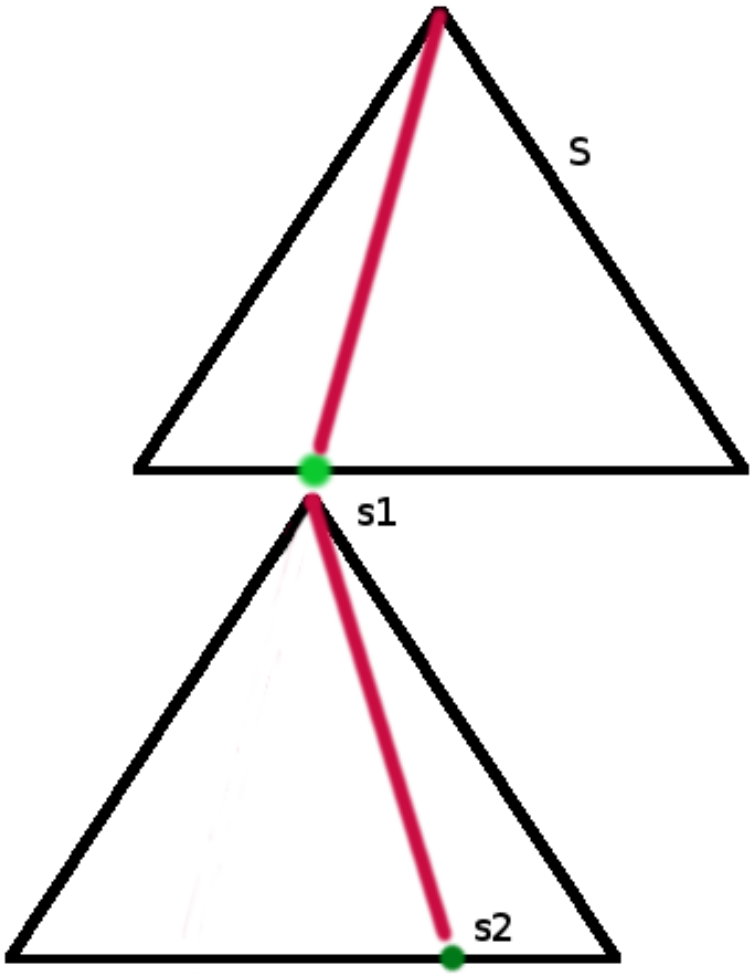


Die Grundidee:



Die Grundidee:





Was wir gerne hätten: Lenses, die sich einfach miteinander kombinieren lassen.

```
composeL :: Lens' s s1
         -> Lens' s1 s2
         -> Lens' s s2
```

Wir wissen bereits, dass Composability ein großer Vorteil für funktioniale Konstrukte ist.

„Puzzle Programming“ macht es uns einfacher, korrekte und elegante Programme zu schreiben.

Aber warum brauchen wir sowas? Geht das nicht alles schon mit Pattern-Matching?

Aber warum brauchen wir sowas? Geht das nicht alles schon mit Pattern-Matching?

```
data Person = Person { name :: String
                     , addr :: Address }

data Address = Address { road :: String
                       , city :: String
                       , pstc :: Int }

setName :: String -> Person -> Person
setName nm p = p { name = nm } -- record update notation

setPostcode :: Int -> Person -> Person
setPostcode pc p = p { addr = addr p { pstc = pc } }
```

Ja, das geht. Aber es wird schnell ermüdend. Wer beim Erklären zu oft „blah-blah“ sagt, sollte sich um elegantere Wege oder Automatisierung bemühen.

Angenommen, wir hätten jetzt eine `Lens` für jedes Feld, ...

```
lname :: Person -> String
laddr :: Person -> Address
```

Angenommen, wir hätten jetzt eine `Lens` für jedes Feld, ...

```
lname :: Person -> String
laddr :: Person -> Address
```

... Funktionen, die `Lenses` zum lesen und schreiben benutzen, ...

```
view :: Lens' s a -> s -> s
set  :: Lens' s a -> a -> s -> s
```

Angenommen, wir hätten jetzt eine Lens für jedes Feld, ...

```
lname :: Person -> String
laddr :: Person -> Address
```

... Funktionen, die Lenses zum lesen und schreiben benutzen, ...

```
view :: Lens' s a -> s -> s
set  :: Lens' s a -> a -> s -> s
```

... dann könnten wir (zusammen mit der composeL-Funktion) deutlich eleganteren und effizienteren Code schreiben:

```
setPostcode :: Int -> Person -> Person
setPostcode pc p = set (laddr `composeL` lpstc) pc p
```

Der naive Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter zu bündeln:

```
data LensR s a = L { view :: s -> a
                   , set  :: a -> s -> s }
```

Der naive Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter zu bündeln:

```
data LensR s a = L { view :: s -> a
                   , set  :: a -> s -> s }
```

Mit etwas Hirnschmalz kriegen wir sogar `composeL`:

```
composeL :: LensR s s1 -> LensR s1 s2 -> LensR s s2
composeL (L v1 u1) (L v2 u2) = L (\s -> v2 (v1 s))
                                 (\a s -> u1 (u2 a (v1 s)) s)
```

Der naive Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter zu bündeln:

```
data LensR s a = L { view :: s -> a
                   , set  :: a -> s -> s }
```

Mit etwas Hirnschmalz kriegen wir sogar `composeL`:

```
composeL :: LensR s s1 -> LensR s1 s2 -> LensR s s2
composeL (L v1 u1) (L v2 u2) = L (\s -> v2 (v1 s))
                                 (\a s -> u1 (u2 a (v1 s)) s)
```

. . . all das ist aber sehr ineffizient. Falls wir `over` haben wollen

```
over :: Lens s a -> (a -> a) -> s -> s
```

. . . müssten wir erst getten, dann setten. *Nicht cool*.

Wir könnten jetzt einfach eine `modify`-Funktion hinzufügen:

```
data LensR s a = L { view   :: s -> a
                   , set    :: a -> s -> s
                   , modify :: (a -> a) -> s -> s }
```

Wir könnten jetzt einfach eine `modify`-Funktion hinzufügen:

```
data LensR s a = L { view   :: s -> a
                   , set    :: a -> s -> s
                   , modify :: (a -> a) -> s -> s }
```

Das Problem dabei ist nur, dass wir sehr schnell zu viele Funktionen haben. Was ist mit effektvollen Veränderungen? Oder mit Veränderungen, die Fehlschlagen können?

Wir könnten jetzt einfach eine `modify`-Funktion hinzufügen:

```
data LensR s a = L { view   :: s -> a
                   , set    :: a -> s -> s
                   , modify :: (a -> a) -> s -> s }
```

Das Problem dabei ist nur, dass wir sehr schnell zu viele Funktionen haben. Was ist mit effektvollen Veränderungen? Oder mit Veränderungen, die Fehlschlagen können?

```
data LensR s a =
    L { view          :: s -> a
      , set           :: a -> s -> s
      , modify        :: (a -> a) -> s -> s
      , modidyIO      :: (a -> IO a) -> s -> IO s
      , modifyMaybe :: (a -> Maybe a) -> s -> Maybe s }
```

Diese Datenstruktur wächst uns schnell über den Kopf und ist dafür nicht mal sehr flexibel.


The End
...OR IS IT?

Das geübte Auge findet zumindest für den letzten Schritt noch einen Ausweg.

Das geübte Auge findet zumindest für den letzten Schritt noch einen Ausweg.
Wir könnten immerhin die Funktionen `modifyMaybe` und `modifyIO` (und alle, die dem gleichen Muster folgen) zusammenfassen:

```
data LensR s a =
    L { view    :: s -> a
      , set     :: a -> s -> s
      , modify  :: (a -> a) -> s -> s
      , modidyF :: Functor f => (a -> f a) -> s -> f s }
```

Und das ist eine wirklich gute Idee.

Edward's big insight:

Edward's big insight:

Eine *noch* bessere Idee ist es allerdings (und das ist die große Idee hinter `Lens`), auch die Funktionen `view`, `set` und `modify` über die Funktion `modifyF` auszudrücken!

Edward's big insight:

Eine *noch* bessere Idee ist es allerdings (und das ist die große Idee hinter `Lens`), auch die Funktionen `view`, `set` und `modify` über die Funktion `modifyF` auszudrücken!

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s
```

Edward's big insight:

Eine *noch* bessere Idee ist es allerdings (und das ist die große Idee hinter `Lens`), auch die Funktionen `view`, `set` und `modify` über die Funktion `modifyF` auszudrücken!

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s
```

Das ist nur noch ein `type`, also ein Alias von einem Typen auf einen anderen. Mehr brauchen wir nicht.

Fun Fact: `Lens'` und `LensR` sind *isomorph*!

Fun Fact: `Lens'` und `LensR` sind *isomorph*!

Das bedeutet wir können folgende Funktionen schreiben:

```
lensR2Lens :: LensR s a -> Lens' s a
lens2LensR :: Lens' s a -> LensR s a
```

Fun Fact: `Lens'` und `LensR` sind *isomorph*!

Das bedeutet wir können folgende Funktionen schreiben:

```
lensR2Lens :: LensR s a -> Lens' s a
lens2LensR :: Lens' s a -> LensR s a
```

Hier ist eine Richtung, wir werden uns aber nicht lange damit aufhalten. Übungsaufgabe. ;-)

```
view :: Lens' s a -> s -> a
view ln = getConst . ln Const

set :: Lens' s a -> a -> s -> s
set ln x = getIdentity . ln (Identity . const x)

-- one way of the isomorphism
lens2LensR :: Lens' s a -> LensR s a
lens2LensR ln = L { viewR = view ln, setR = set ln }
```

Bisher haben wir uns nur angeschaut, wie wir `Lenses` benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Bisher haben wir uns nur angeschaut, wie wir `Lenses` benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s
```

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}
```

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}


-- name :: Functor f => (String -> f String)
--                   -> Person  -> f Person
name :: Lens' Person String
.
```

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}


-- name :: Functor f => (String -> f String)
--                   -> Person  -> f Person
name :: Lens' Person String
name fn (P n b) = ... ähm ...
```

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}


-- name :: Functor f => (String -> f String)
--                   -> Person  -> f Person
name :: Lens' Person String
name fn (P n b) = fmap (\n' -> P n' b) (fn n)
```

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

```
type Lens' = forall f. Functor f => (a -> f a) -> s - > f s

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}


-- name :: Functor f => (String -> f String)
--                   -> Person  -> f Person
name :: Lens' Person String
name fn (P n b) = fmap (\n' -> P n' b) (fn n)
```

Mit etwas mentaler Gymnastik merken wir: Die Typen stimmen!

QuickCheck
(Randomised Property-Based Testing)