Práce se složitějšími datovými strukturami (*zippers*, *lenses*)

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

Motivace I.

Příklad: chceme naprogramovat funkci

```
findContext :: (a -> Bool) -> Int -> [a] -> Maybe [a]
```

kde `findContext p n l` bude vyhledávat v seznamu `l` hodnotu splňující predikát `p`, ale vrátí nejen nalezenou hodnotu, ale navíc i `n` hodnot před a po této hodnotě.

```
findContext even 2 [1,3,5,6,7,9,11,13]
    ⇝* Just [3,5,6,7,9]
```

Motivace II.

```haskell
import Data.List (findIndex)

findContext1 :: (a->Bool) -> Int -> [a] -> Maybe [a]
findContext1 p n xs = do
  index <- findIndex p xs
  return . drop (index - n) $ take (index + n + 1) xs
```

Motivace II.

```
import Data.List (findIndex)

findContext1 :: (a->Bool) -> Int -> [a] -> Maybe [a]
findContext1 p n xs = do
  index <- findIndex p xs
  return . drop (index - n) $ take (index + n + 1) xs
```

Jak to naprogramovat tak, abychom seznam `xs` neprocházeli zbytečně vícekrát?

Motivace III.

```
findContext2 :: (a->Bool) -> Int -> [a] -> Maybe [a]
findContext2 p n xs = fn xs []
  where
    fn [] _ = Nothing
    fn (x:xs) back
        | p x = prepend (take n back) (x : take n xs)
        | otherwise = fn xs (x : back)
    prepend [] xs = Just xs
    prepend (b:bs) xs = prepend bs (b:xs)
```

Seznamy

Co když budeme chtít jinou funkci na seznamech, která pracuje s lokálním kontextem velkého seznamu?

Seznamy

Co když budeme chtít jinou funkci na seznamech, která pracuje s lokálním kontextem velkého seznamu?

Zevšeobecníme kontextové procházení seznamů:

```
data LZipper a = LZip [a] [a]

goForward  :: LZipper a -> Maybe (LZipper a)
goBackward :: LZipper a -> Maybe (LZipper a)
modifyLZip :: (a -> a) -> LZipper a -> LZipper a

listToZip :: [a] -> LZipper a
zipToList :: LZipper a -> [a]
```

Binární stromy I.

Jak si pamatovat pozici v binárním stromu, abychom mohli efektivně zpracovávat okolí aktuálního uzlu?

```
data BinTree a = BNode (BinTree a) a (BinTree a)
               | BEmpty
```

Binární stromy I.

Jak si pamatovat pozici v binárním stromu, abychom mohli efektivně zpracovávat okolí aktuálního uzlu?

```
data BinTree a = BNode (BinTree a) a (BinTree a)
               | BEmpty
```

Můžeme si pamatovat trasu k upravovanému uzlu

```
data Direction = L | R
modify :: BinTree a -> [Direction] -> a -> BinTree a
```

Neefektivní při opakovaných úpravách, úpravách blízkých uzlů!

Binární stromy II.

Ve stromě se budeme pohybovat, ale zároveň si budeme pamatovat i trasu zpět pro rekonstrukci stromu.

```
data BinTree a = BNode (BinTree a) a (BinTree a)
               | BEmpty
```

Binární stromy II.

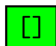
Ve stromě se budeme pohybovat, ale zároveň si budeme pamatovat i trasu zpět pro rekonstrukci stromu.

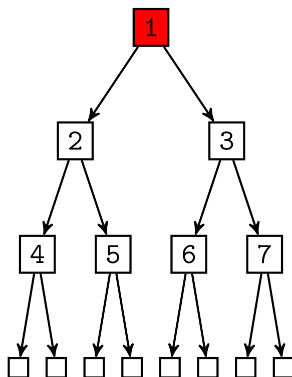
```
data BinTree a = BNode (BinTree a) a (BinTree a)
               | BEmpty


data TreeDir a = TLeft a (BinTree a)
               | TRight (BinTree a) a
               deriving ( Eq, Show, Read )

data TreeZipper a = TZip [TreeDir a] (BinTree a)
                  deriving ( Eq, Show, Read )
```

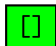
Binární stromy: příklad

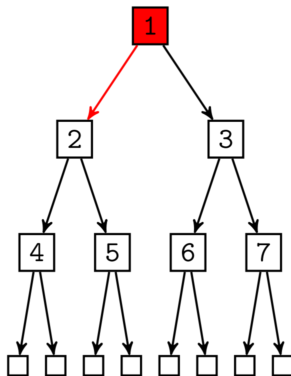





```
zipper = TZip [] (N (N (N E 4 E) 2 (N E 5 E) 1 (N (N
E 6 E) 3 (N E 7 E)))
```

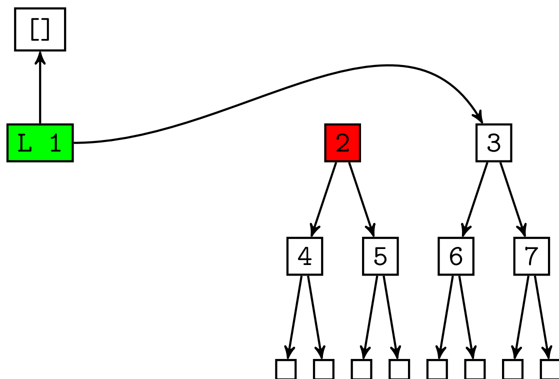
Binární stromy: příklad





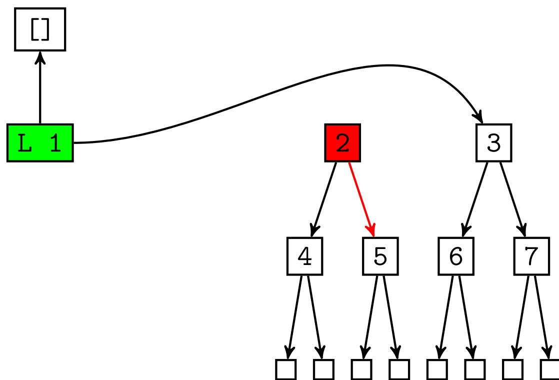

```
goLeft zipper
```

Binární stromy: příklad



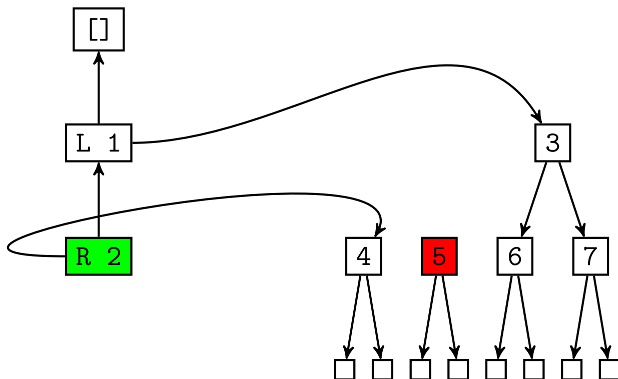

```
zipper TZip [L 1 (N (N E 6 E) 3 (N E 7 E))] (N (N E 4
 E) 2 (N E 5 E))
```

Binární stromy: příklad



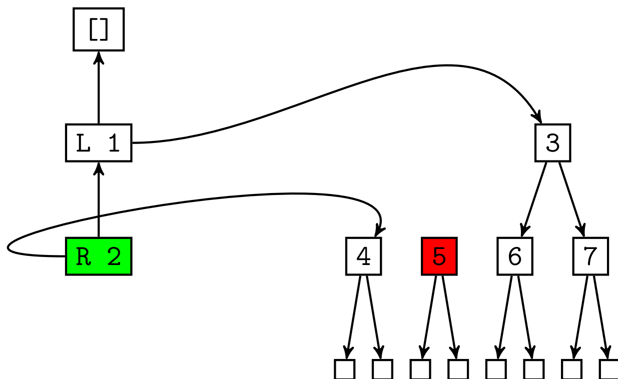

```
goRight zipper
```

Binární stromy: příklad



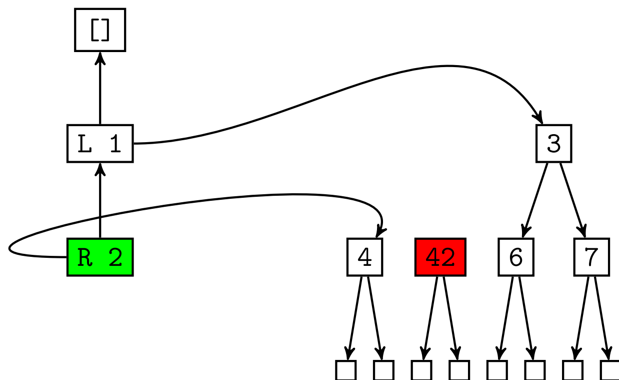

```
zipper TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N E 7
 E))] (N E 5 E)
```

Binární stromy: příklad



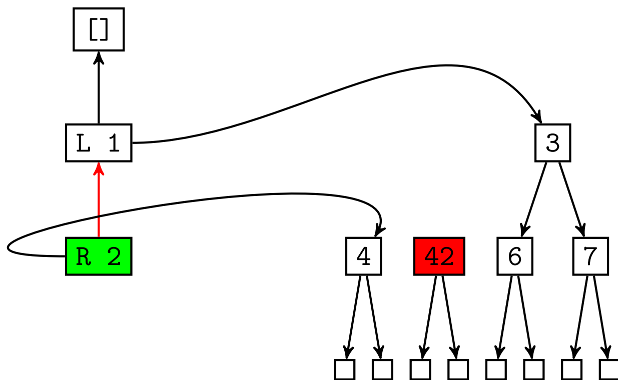

```
modify (+ 37) zipper
```

Binární stromy: příklad



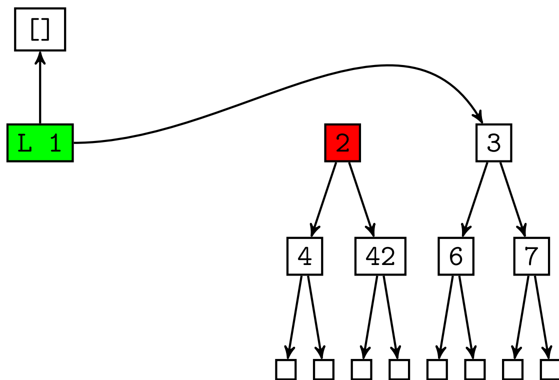

```
zipper = TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N
E 7 E))] (N E 42 E)
```

Binární stromy: příklad




```
goUp zipper
```

Binární stromy: příklad




```
zipper TZip [L 1 (N (N E 6 E) 3 (N E 7 E))] (N (N E 4
 E) 2 (N E 42 E))
```

Binární stromy: implementace

Manipulaci můžeme realizovat například pomocí:

```
goLeft  :: TZipper a -> TZipper a
goRight :: TZipper a -> TZipper a
goUp    :: TZipper a -> TZipper a
modify  :: (a -> a) -> TZipper a -> TZipper a
```

Zippers: Shrnutí

zipper pro danou datovou strukturu je datová struktura obohacená o „kontext“, který umožňuje efektivně manipulovat pozici ve struktuře

- **seznam**:
  zipper je dvojice seznamů: jeden obsahuje dosud neprojitý seznam, druhý prvky, které již byly projity v opačném pořadí
- **binární strom**:
  zipper je aktuální strom spolu se seznamem kroků zpět ke kořeni (vrchol + levý nebo pravý podstrom)
- obdobně pro složitější struktury

Lenses: Motivační příklad

```haskell
data Person = Person {
    eid :: Int ,
    name :: String } deriving Show

data Project = Project {
    lead :: Person ,
    techLead :: Person } deriving Show

data Team = Team {
    job1 :: Project ,
    job2 :: Project } deriving Show
```

Porovnání s jinými jazyky

```haskell
data Person  = Person  { eid :: Int, name :: String }
data Project = Project { lead :: Person, techLead :: Person }
data Team    = Team    { job1 :: Project, job2 :: Project }
```

Jak změnit EID vedoucího prvního projektu v týmu `t`?

Porovnání s jinými jazyky

```
data Person  = Person  { eid :: Int, name :: String }
data Project = Project { lead :: Person, techLead :: Person }
data Team    = Team    { job1 :: Project, job2 :: Project }
```

Jak změnit EID vedoucího prvního projektu v týmu `t`?

V Haskellu (využívajíc záznamů):

- ```
  t { job1 = (job1 t) { lead =
        (lead $ job1 t) { eid = 123 }
    } }
  ```
  ```

Porovnání s jinými jazyky

```haskell
data Person  = Person  { eid :: Int, name :: String }
data Project = Project { lead :: Person, techLead :: Person }
data Team    = Team    { job1 :: Project, job2 :: Project }
```

Jak změnit EID vedoucího prvního projektu v týmu `t`?

V Haskellu (využívajíc záznamů):

- ```haskell
  t { job1 = (job1 t) { lead =
          (lead $ job1 t) { eid = 123 }
      } }
  ```

V „běžných“ jazycích (C, Java, ...):

- `t.job1.lead.eid = 123;`
  ```

Datové struktury: setters & updaters

```
data Person  = Person  { eid :: Int, name :: String }
data Project = Project { lead :: Person, techLead :: Person }
data Team    = Team    { job1 :: Project, job2 :: Project }
```

Napišme si „setter“ a „updater“:

```
setEid :: Int -> Person -> Person
overLead :: (Person -> Person) -> Project -> Project
overJob1 :: (Project -> Project) -> Team -> Team
```

Datové struktury: setters & updaters

```haskell
data Person  = Person  { eid :: Int, name :: String }
data Project = Project { lead :: Person, techLead :: Person }
data Team    = Team    { job1 :: Project, job2 :: Project }
```

Napišme si „setter“ a „updater“:

```haskell
setEid :: Int -> Person -> Person
overLead :: (Person -> Person) -> Project -> Project
overJob1 :: (Project -> Project) -> Team -> Team
```

Změna EID vedoucího prvního projektu v týmu `t`:

- `(overJob1 . overLead . setEid) 123 t`
- Pozorování: Modifikační funkce se dobře skládají.

Lenses: základní myšlenka

lens $\approx$ kombinace „getter“ a „setter“ funkcí pro manipulaci s (pod)částí dané datové struktury

- lze je skládat, brát jako argumenty, modifikovat, vracet, ...
- ulehčují práci se složitými (vnorenými) datovými strukturami

Lenses: základní myšlenka

lens ≈ kombinace „getter“ a „setter“ funkcí pro manipulaci s (pod)částí dané datové struktury

- lze je skládat, brát jako argumenty, modifikovat, vracet, ...
- ulehčují práci se složitými (vnorenými) datovými strukturami

První (naivní) implementace:

```
data Lens'' s a = Lens''
    { view ::              s -> a
    , set  ::        a  -> s -> s
    , over :: (a -> a) -> s -> s
    }
```

Příklad

```
data Person = Person { eid :: Int, name :: String }
data Lens'' s a = Lens''
    { view ::              s -> a
    , set  ::        a  -> s -> s
    , over :: (a -> a) -> s -> s }
```

Jak vypadá *lens* pro `eid` v rámci `Person`?

Příklad

```
data Person = Person { eid :: Int, name :: String }
data Lens'' s a = Lens''
    { view ::              s -> a
    , set  ::        a  -> s -> s
    , over :: (a -> a) -> s -> s }
```

Jak vypadá *lens* pro eid v rámci Person?

```
eidLens :: Lens'' Person Int
eidLens = Lens''
    { view = ( \       p -> eid p )
    , set  = ( \neweid p -> p { eid = neweid } )
    , over = ( \f      p -> p { eid = f (eid p) } ) }
```

Příklad

```haskell
data Person = Person { eid :: Int, name :: String }
data Lens'' s a = Lens''
    { view ::                s -> a
    , set  ::          a  -> s -> s
    , over :: (a -> a) -> s -> s }
```

Jak vypadá *lens* pro eid v rámci Person?

```haskell
eidLens :: Lens'' Person Int
eidLens = Lens''
    { view = ( \        p -> eid p )
    , set  = ( \neweid p -> p { eid = neweid } )
    , over = ( \f       p -> p { eid = f (eid p) } ) }
```

Použití:

- `view eidLens       p`
- `set  eidLens 123   p`
- `over eidLens (+1)  p`

A co vedlejší efekty?

A co když bude úprava mít vedlejší efekty?

- `adjustEidWithCheck :: Int -> IO Int`
  (kontroluje platnost daného EID v databázi)

A co vedlejší efekty?

A co když bude úprava mít vedlejší efekty?

- `adjustEidWithCheck :: Int -> IO Int`
  (kontroluje platnost daného EID v databázi)

Přidejme ještě „updater“ tvořící výsledek ve funktoru.

- `overF :: Functor f => (a -> f a) -> s -> f s`

A co vedlejší efekty?

A co když bude úprava mít vedlejší efekty?

- `adjustEidWithCheck :: Int -> IO Int`
  (kontroluje platnost daného EID v databázi)

Přidejme ještě „updater“ tvořící výsledek ve funktoru.

- `overF :: Functor f => (a -> f a) -> s -> f s`

Pro `eidLens :: Lens'' Person Int`

```
overF :: Functor f =>
    (Int -> f Int) -> Person -> f Person
overF f p = fmap
    (\neweid -> p { eid = neweid })
    (f $ eid p)
```

Zobecňování I.

```
data Lens'' s a = Lens''
    { view  ::                              s ->   a
    , set   ::                        a  -> s ->   s
    , over  ::                 (a ->   a) -> s ->   s
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Naše *lens* začíná mít příliš mnoho operátorů ...
Nemohli bychom vyjářit `set` pomocí ostatních částí `Lens''`?

Zobecňování I.

```
data Lens'' s a = Lens''
    { view  ::                                  s ->   a
    , set   ::                           a   -> s ->   s
    , over  ::                   (a ->   a) -> s ->   s
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Naše *lens* začíná mít příliš mnoho operátorů ...
Nemohli bychom vyjářit `set` pomocí ostatních částí `Lens''`?

```
set :: a -> s -> s
set newval object = over (\_ -> newval) object
```

Zobecňování II.

```
data Lens'' s a = Lens''
    { view  ::                               s ->   a
    , over  ::                   (a ->   a) -> s ->   s
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Nemohli bychom vyjářit `over` pomocí ostatních částí `Lens''`?

Zobecňování II.

```
data Lens'' s a = Lens''
    { view  ::                              s ->   a
    , over  ::                (a ->   a) -> s ->   s
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Nemohli bychom vyjářit `over` pomocí ostatních částí `Lens''`?

Knihovní funktor `Identity`:

```
newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
  fmap f (Identity x) = Identity (f x)
```

Zobecňování II.

```
data Lens'' s a = Lens''
    { view  ::                            s ->   a
    , over  ::                 (a ->   a) -> s ->   s
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Nemohli bychom vyjářit `over` pomocí ostatních částí `Lens''`?

Knihovní funktor `Identity`:

```
newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
  fmap f (Identity x) = Identity (f x)
```

```
over :: (a -> a) -> s -> s
over f object = runIdentity $ overF (Identity . f) s
```

Zobecňování III.

```
data Lens'' s a = Lens''
    { view  ::                                 s ->   a
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Nemohli bychom vyjářit `view` pomocí ostatních částí `Lens''`?

Zobecňování III.

```
data Lens'' s a = Lens''
    { view  ::                              s ->   a
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Nemohli bychom vyjářit `view` pomocí ostatních částí `Lens''`?

Knihovní funktor `Const`:

```
newtype Const a b = Const { getConst :: a }
instance Functor (Const a) where
  fmap _ (Const a) = Const a
```

Zobecňování III.

```
data Lens'' s a = Lens''
    { view  ::                                s ->   a
    , overF :: Functor f => (a -> f a) -> s -> f s }
```

Nemohli bychom vyjářit `view` pomocí ostatních částí `Lens''`?

Knihovní funktor `Const`:

```
newtype Const a b = Const { getConst :: a }
instance Functor (Const a) where
  fmap _ (Const a) = Const a
```

```
view :: s -> a
view s = getConst $ overF Const s
```

Příklad

```
data Person = Person { eid :: Int, name :: String }
eidLens :: Lens'' Person Int
eidLens = Lens'' {overF = \f p -> fmap (\x -> p {eid=x}) (f $ eid p)}
newtype Const a b = Const { getConst :: a }
fmap _ (Const a) = Const a
view s = getConst $ overF Const s
```

```
viewEid p
    where p = (Person 123 "John Doe")
```

Příklad

```
data Person = Person { eid :: Int, name :: String }
eidLens :: Lens'' Person Int
eidLens = Lens'' {overF = \f p -> fmap (\x -> p {eid=x}) (f $ eid p)}
newtype Const a b = Const { getConst :: a }
fmap _ (Const a) = Const a
view s = getConst $ overF Const s
```

```
viewEid p
    where p = (Person 123 "John Doe")
```

$\rightsquigarrow$ `getConst $ (overF eidLens) Const p`

Příklad

```
data Person = Person { eid :: Int, name :: String }
eidLens :: Lens'' Person Int
eidLens = Lens'' {overF = \f p -> fmap (\x -> p {eid=x}) (f $ eid p)}
newtype Const a b = Const { getConst :: a }
fmap _ (Const a) = Const a
view s = getConst $ overF Const s
```

```
viewEid p
    where p = (Person 123 "John Doe")
```

```
⇝ getConst $ (overF eidLens) Const p
⇝ getConst $ fmap (\x -> p {eid=x}) (Const $ eid p)
```

Příklad

```haskell
data Person = Person { eid :: Int, name :: String }
eidLens :: Lens'' Person Int
eidLens = Lens'' {overF = \f p -> fmap (\x -> p {eid=x}) (f $ eid p)}
newtype Const a b = Const { getConst :: a }
fmap _ (Const a) = Const a
view s = getConst $ overF Const s
```

```haskell
viewEid p
    where p = (Person 123 "John Doe")
```

$\rightsquigarrow$ `getConst $ (overF eidLens) Const p`

$\rightsquigarrow$ `getConst $ fmap (\x -> p {eid=x}) (Const $ eid p)`

$\rightsquigarrow$ `getConst $ fmap (\x -> p {eid=x}) (Const 123)`

Příklad

```
data Person = Person { eid :: Int, name :: String }
eidLens :: Lens'' Person Int
eidLens = Lens'' {overF = \f p -> fmap (\x -> p {eid=x}) (f $ eid p)}
newtype Const a b = Const { getConst :: a }
fmap _ (Const a) = Const a
view s = getConst $ overF Const s
```

```
viewEid p
    where p = (Person 123 "John Doe")
```

```
⇝ getConst $ (overF eidLens) Const p
⇝ getConst $ fmap (\x -> p {eid=x}) (Const $ eid p)
⇝ getConst $ fmap (\x -> p {eid=x}) (Const 123)
⇝ getConst $ Const 123
⇝ 123
```

Balík lens

Knihovní balík lens:

- nejznámější Haskellová implementace *lenses*
- *„batteries included“* ⇒ mnoho funkcí i závislostí
- moduly `Control.Lens.*`
- obecnější, než naše *lenses*
- umožňuje generovat *lenses* automaticky

Balík lens

Knihovní balík lens:

- nejznámější Haskellová implementace *lenses*
- *„batteries included“* ⇒ mnoho funkcí i závislostí
- moduly `Control.Lens.*`
- obecnější, než naše *lenses*
- umožňuje generovat *lenses* automaticky

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

view :: Lens' s a ->            s -> a
over :: Lens' s a -> (a -> a) -> s -> s
set  :: Lens' s a ->      a   -> s -> s
set lens b = over lens (\_ -> b)
```

lens: Ukázka použití

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens
data Person  = Person  { _eid :: Int, _name :: String }
data Project = Project { _lead :: Person, _techLead :: Person }
data Team    = Team    { _job1 :: Project, _job2 :: Project }
makeLenses ''Person
makeLenses ''Project
makeLenses ''Team
```

lens: Ukázka použití

```haskell
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens
data Person  = Person  { _eid :: Int, _name :: String }
data Project = Project { _lead :: Person, _techLead :: Person }
data Team    = Team    { _job1 :: Project, _job2 :: Project }
makeLenses ''Person
makeLenses ''Project
makeLenses ''Team
```

```haskell
getJob1LeadEid :: Team -> Int
getJob1LeadEid t = view (job1 . lead . eid) t

setJob1LeadEid :: Int -> Team -> Team
setJob1LeadEid x t = set (job1 . lead . eid) x t
```

lens: Ukázka použití

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens
data Person  = Person  { _eid :: Int, _name :: String }
data Project = Project { _lead :: Person, _techLead :: Person }
data Team    = Team    { _job1 :: Project, _job2 :: Project }
makeLenses ''Person
makeLenses ''Project
makeLenses ''Team

getJob1LeadEid :: Team -> Int
getJob1LeadEid t = view (job1 . lead . eid) t

setJob1LeadEid :: Int -> Team -> Team
setJob1LeadEid x t = set (job1 . lead . eid) x t

getJob1LeadEid' t = t^.job1.lead.eid
setJob1LeadEid' x = job1.lead.eid.~x
```

Lenses jsou jen začátek...

Implementace z balíka *lens*

- *lenses* mohou měnit strukturu objektu:

```
type Lens  s t a b = Functor f => (a -> f b) -> s -> f t
```

- existují pravidla pro validní *lenses*
- možnost pracovat i s vícero hodnotami zaráz (*traversals*)
- možnost pracovat s typy, které mají víc konstruktorů (*prisms*)
- možnost využívat vztahy mezi typy (*isos*)
- ...

Lenses jsou jen začátek...

Implementace z balíka *lens*

- *lenses* mohou měnit strukturu objektu:

```
type Lens  s t a b = Functor f => (a -> f b) -> s -> f t
```

- existují pravidla pro validní *lenses*
- možnost pracovat i s vícero hodnotami zaráz (*traversals*)
- možnost pracovat s typy, které mají víc konstruktorů (*prisms*)
- možnost využívat vztahy mezi typy (*isos*)
- ...

Další čtení v případě zájmu:

- tutoriál na Jakub Arnold Blog
- balík Control.Lens.Tutorial

Zippers, *lenses*: Samostatná práce

- Princip *zippers* můžeme zobecnit typovou (konstruktorovou) třídou. Zamyslete se, jak by deklarace/instance této třidy vypadaly a pak se podívejte na naši implementaci v ISu.
- Podívejte se na větší program, který používá *lenses*: jednoduchou implementaci hry Pong od Edwarda Kmetta přímo v balíku *lens*.
- Přečtěte si motivaci a detaily k třídám Foldable a Traversable a pak tutoriál k obecnějším *lens* na blogu Jakuba Arnolda.