Zippers

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2017

Motivace I.

Příklad: chceme naprogramovat funkci

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

kde `findContext p n l` bude vyhledávat s 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 l = do
  index <- findIndex p l
  return . drop (index - n) $ take (index + n + 1) l
```

Motivace II.

```
import Data.List (findIndex)

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

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

Motivace III.

```
findContext2 :: (a->Bool) -> Int -> [a] -> Maybe [a]
findContext2 p n l = fn l []
  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 seznamy:

```
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.

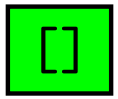
Ve stromě se budeme pohybovat, ale zároveň si budeme pamatovat i trasu zpět pro rekonstrukci stromu.

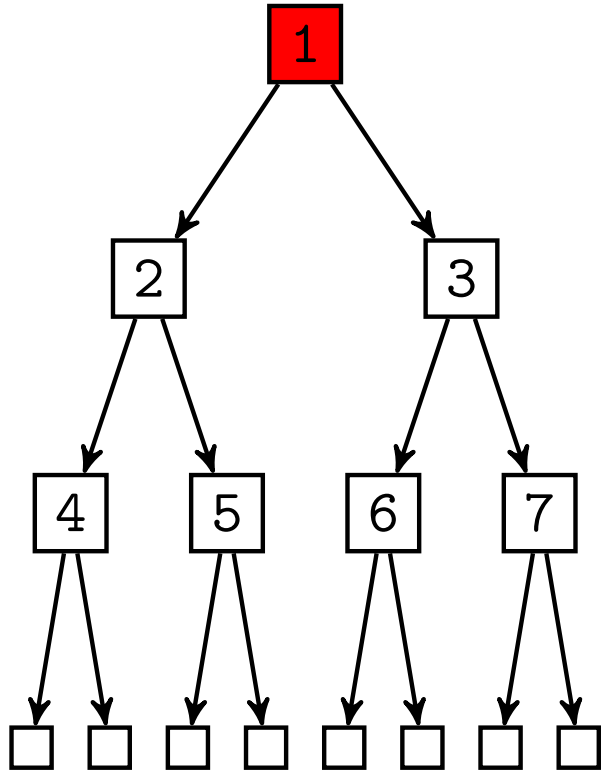
```
data BinTree a = BNode (BinTree a) a (BinTree a)
               | BEmpty


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 )
```

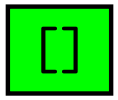
Binární stromy: příklad

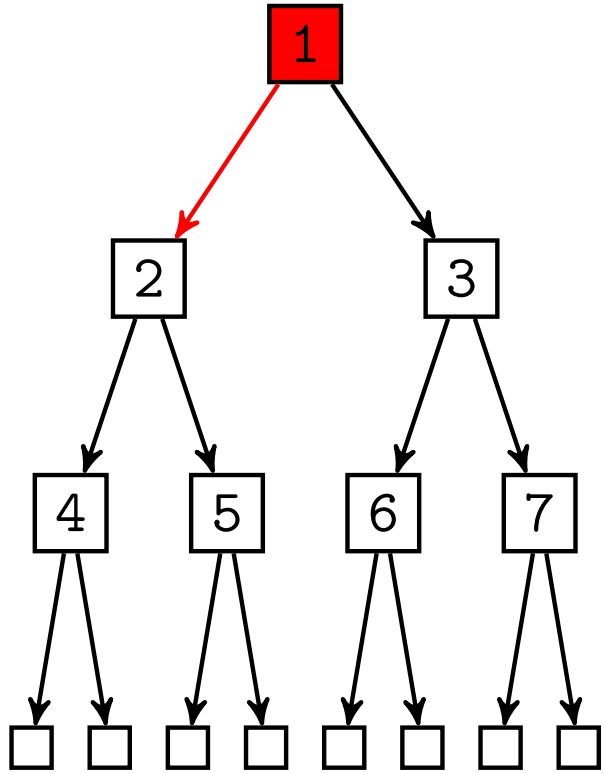





```
zipper = TZip [] (N (N (N E 4 E) 2 (N E 5 E) 1 (N (N
E 6 E) 3 (N E 7 E)))
```

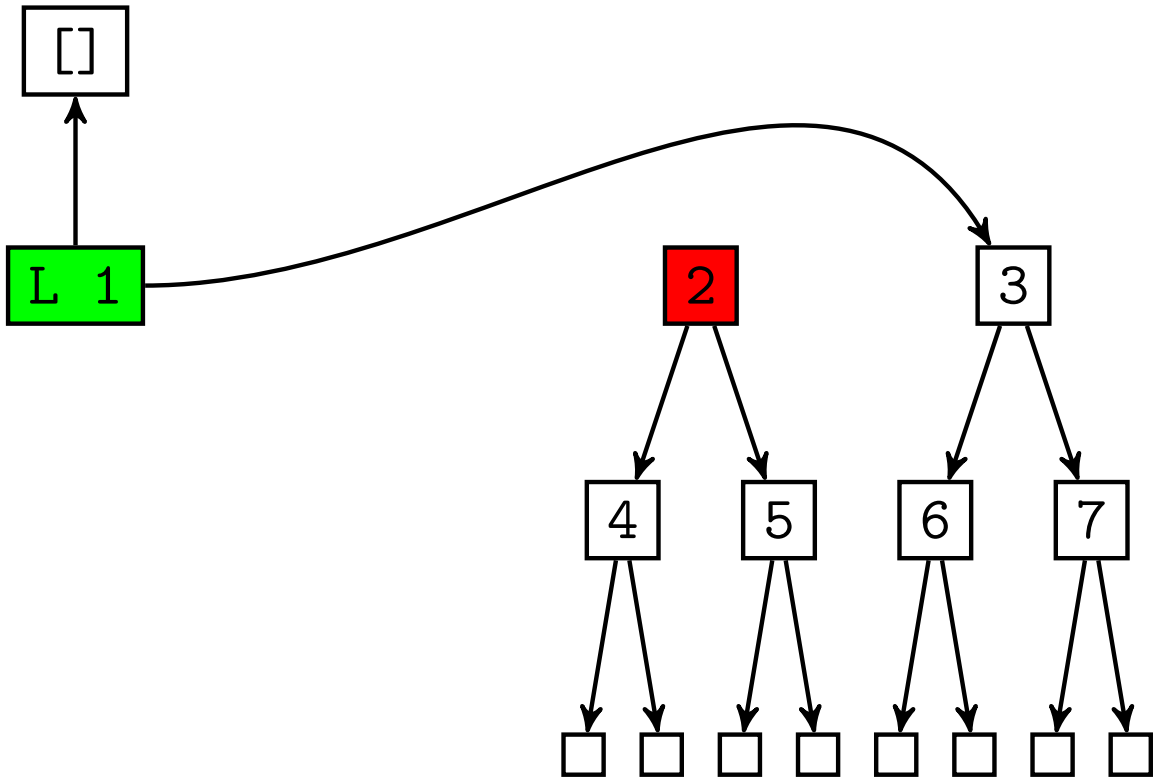
Binární stromy: příklad





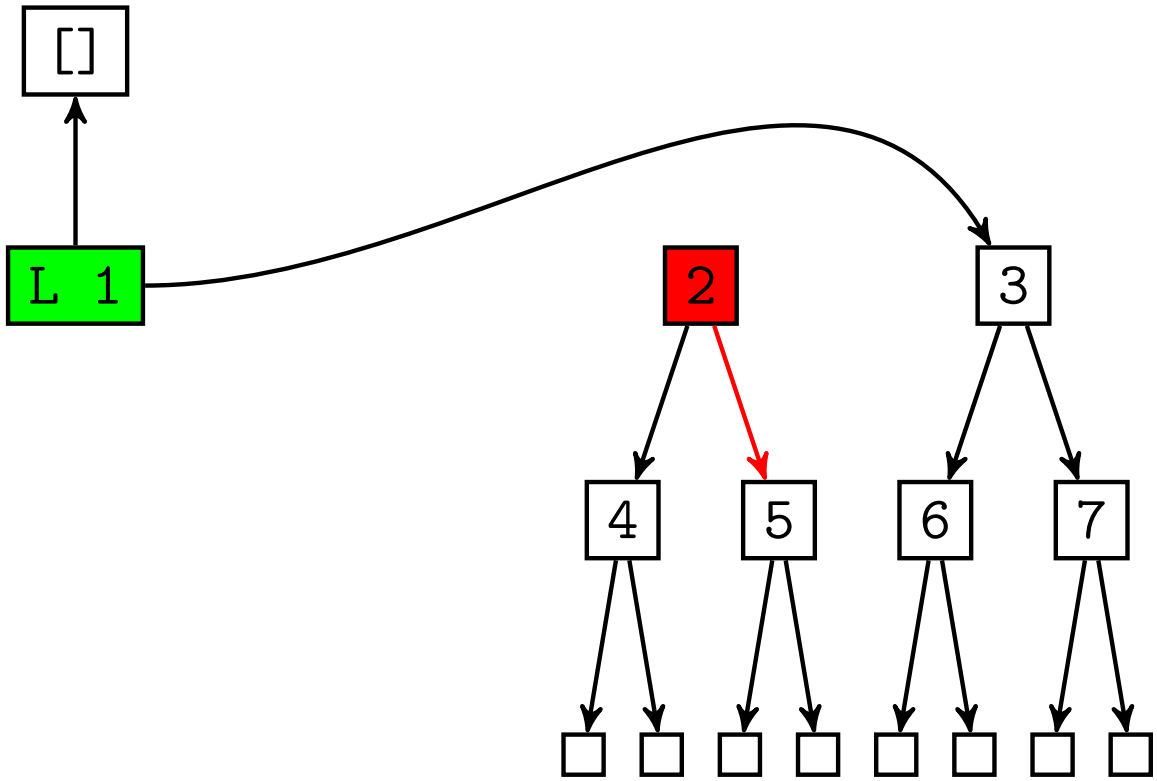

```
goLeft zipper
```

Binární stromy: příklad



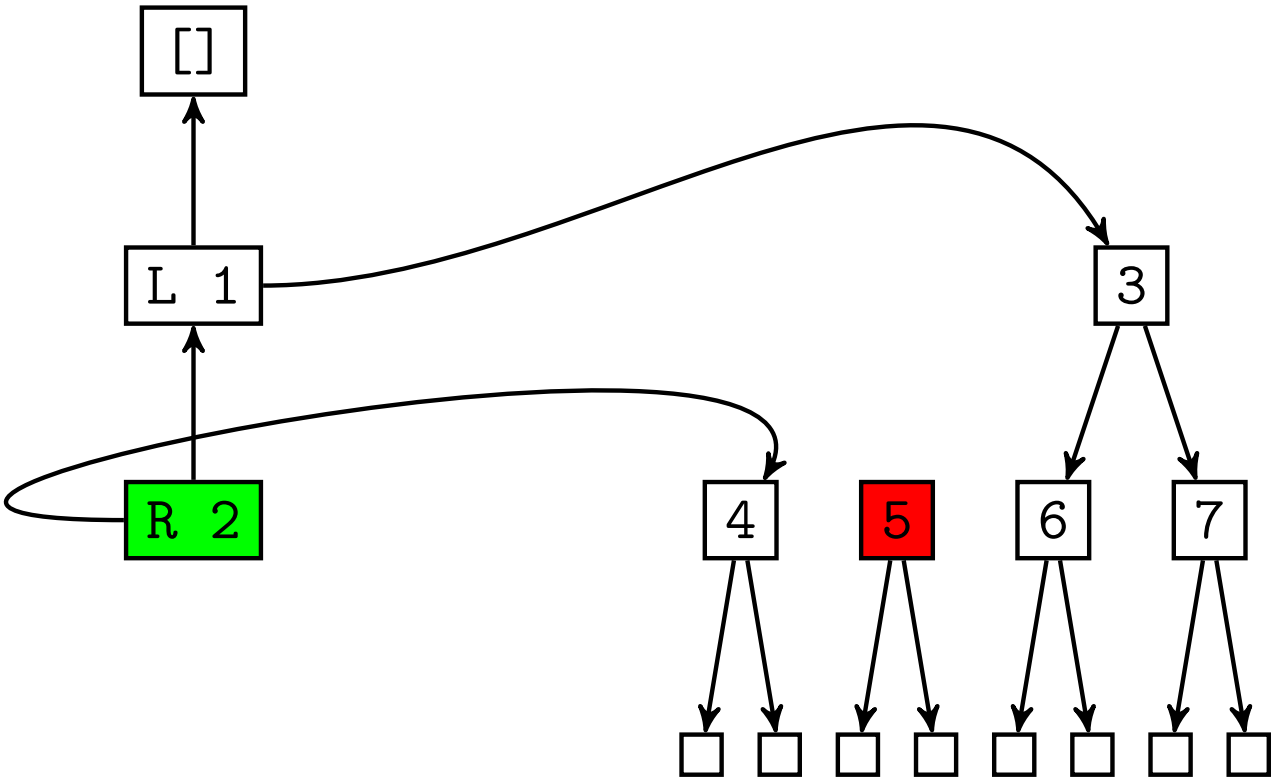

```
zipper TZip [L 1 (N (N E 6 E) 3 (N E 7 E))] (N (N E 4
 E) 2 (N E 5 E))
```

Binární stromy: příklad



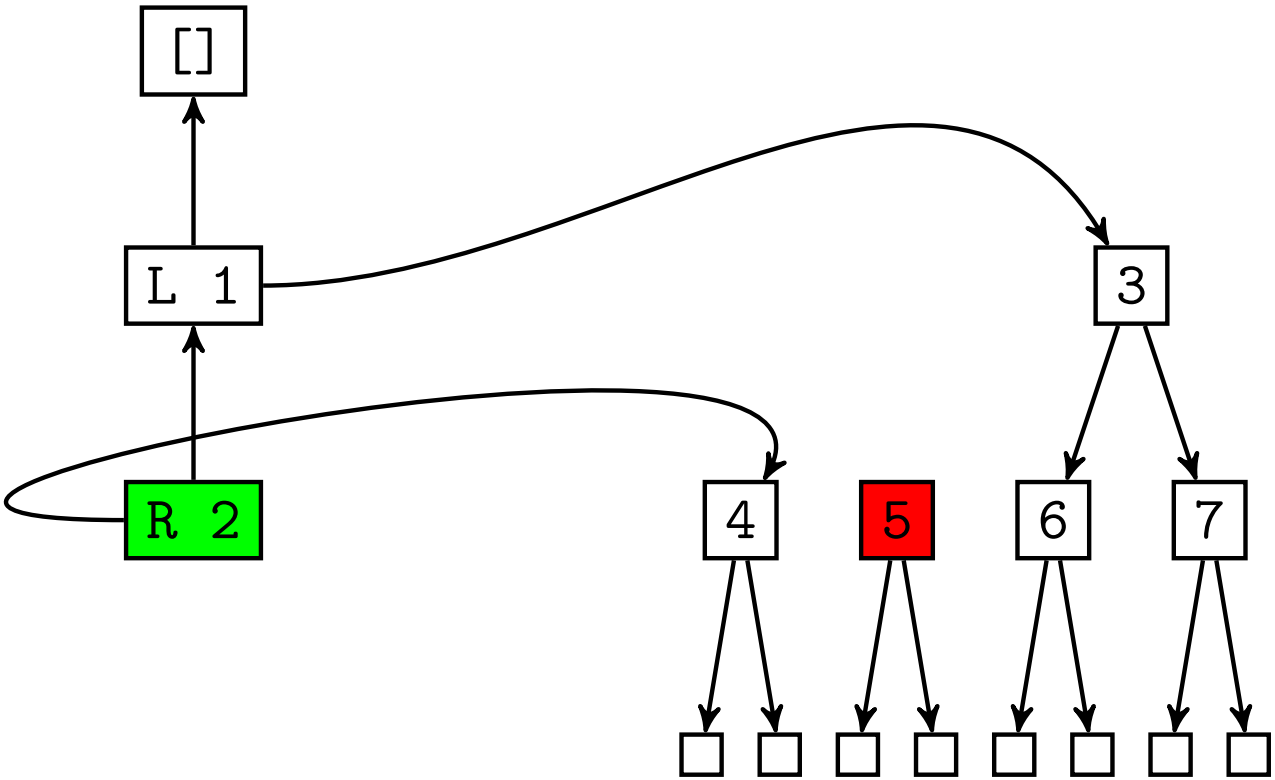

```
goRight zipper
```

Binární stromy: příklad



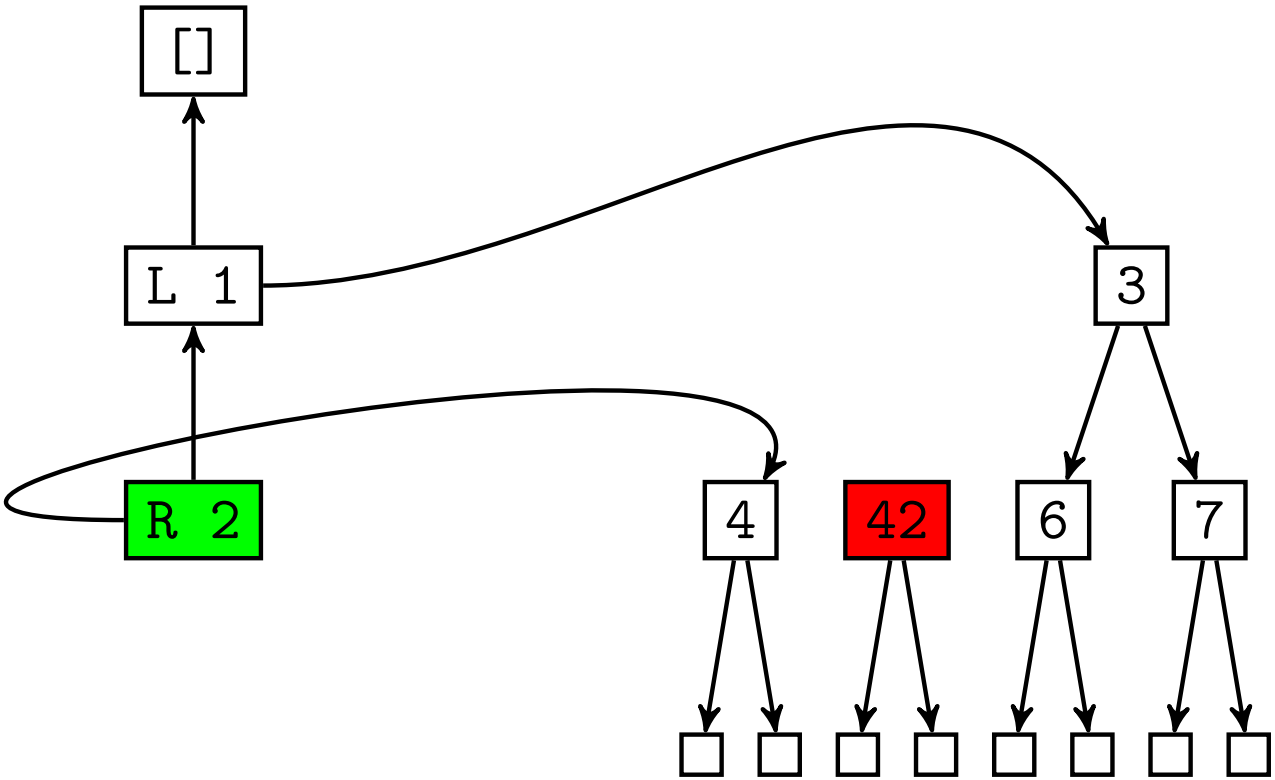

```
zipper TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N E 7
 E))] (N E 5 E)
```

Binární stromy: příklad



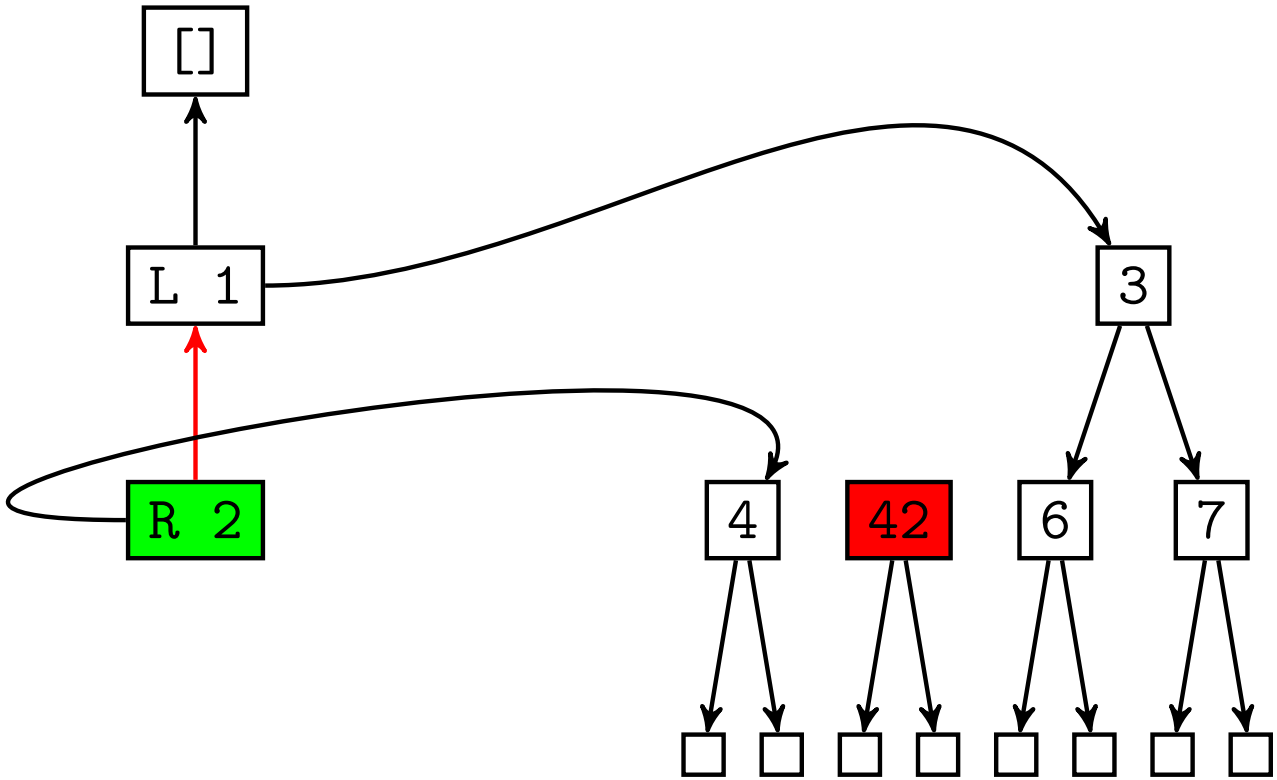

```
modify (+ 37) zipper
```

Binární stromy: příklad



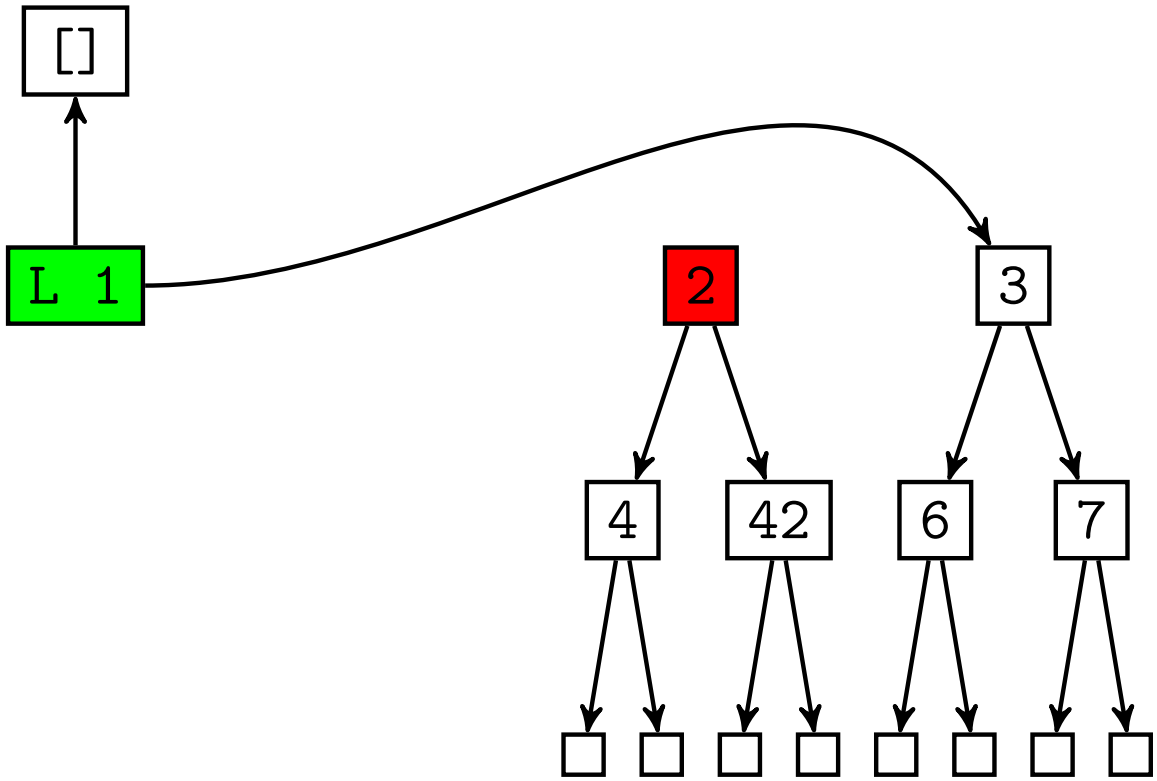

```
zipper = TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N
E 7 E))] (N E 42 E)
```

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

- obecně 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

Zippers

- obecně 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

- pricip zipprů můžeme zobecnit typovou (konstruktorovou) třídou... (viz soubor v ISu)