Zippers

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2015

Motivace

Příklad: chceme naprogramovat funkci

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

kde `findWithContext 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.

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

Motivace

Příklad: chceme naprogramovat funkci

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

kde `findWithContext 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.

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

```
findWithContext 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)
```

Stromy

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

Jak efektivně procházet binární strom a pamatovat si pozici v něm?

Stromy

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

Jak efektivně procházet binární strom a pamatovat si pozici v něm?

- pamatovat si trasu k upravovanému uzlu

```
data Direction = L | R
               deriving ( Eq, Ord, Show, Read )
type Trace = [Direction]
modify :: BinTree a -> Trace -> a -> BinTree a
```

Stromy

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

Jak efektivně procházet binární strom a pamatovat si pozici v něm?

- pamatovat si trasu k upravovanému uzlu

```
data Direction = L | R
               deriving ( Eq, Ord, Show, Read )
type Trace = [Direction]
modify :: BinTree a -> Trace -> a -> BinTree a
```

- neefektivní při opakovaných úpravách, úpravách blízkých uzlů!

Stromy

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 )
```

Stromy

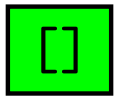
Ve stromě se budeme pohybovat, ale zároveň si budeme pamatovat i trasu zpět pro rekonstrukci stromu.

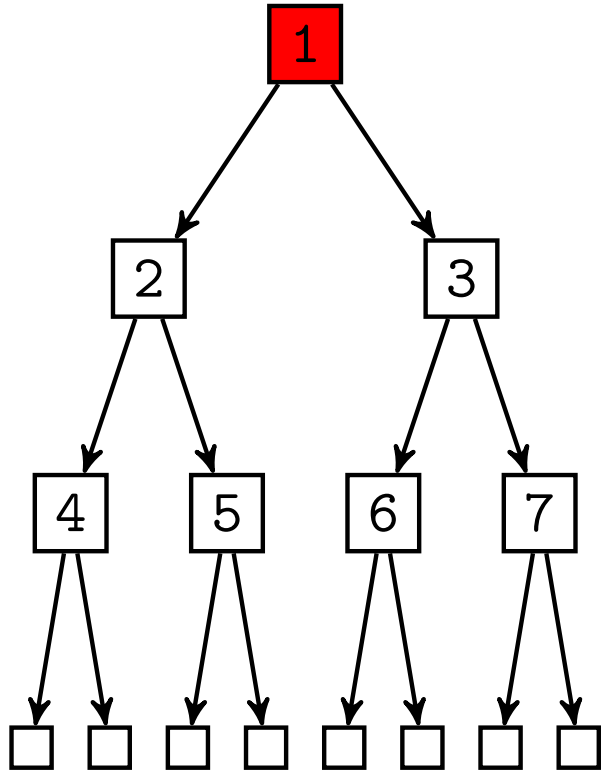
```
data BinTree a = BNode (BinTree a) a (BinTree a)
               | BEmpty

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 )
```

Manipulaci můžeme realizovat například pomocí:

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

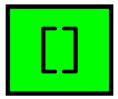
Stromy: příklad

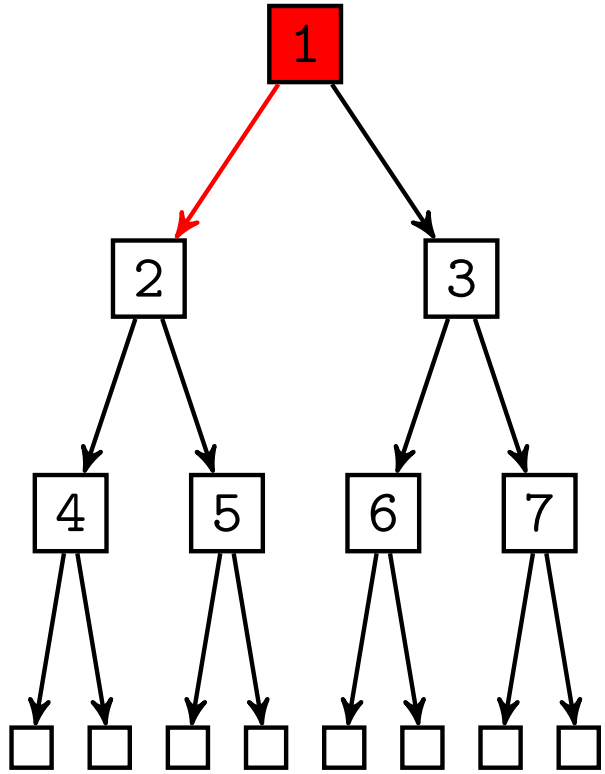





```
zipper = TZip [] (N (N (N E 4 E) 2 (N E 5 E) 1 (N (N
E 6 E) 3 (N E 7 E)))
```

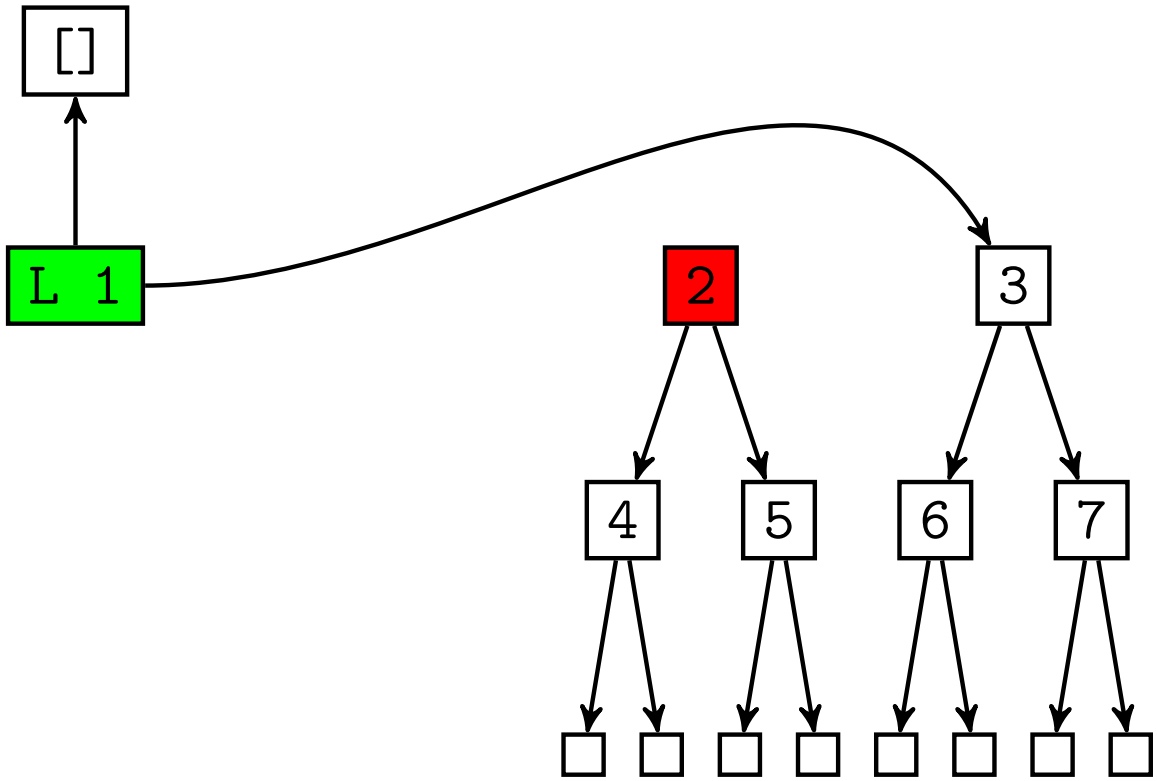
Stromy: příklad





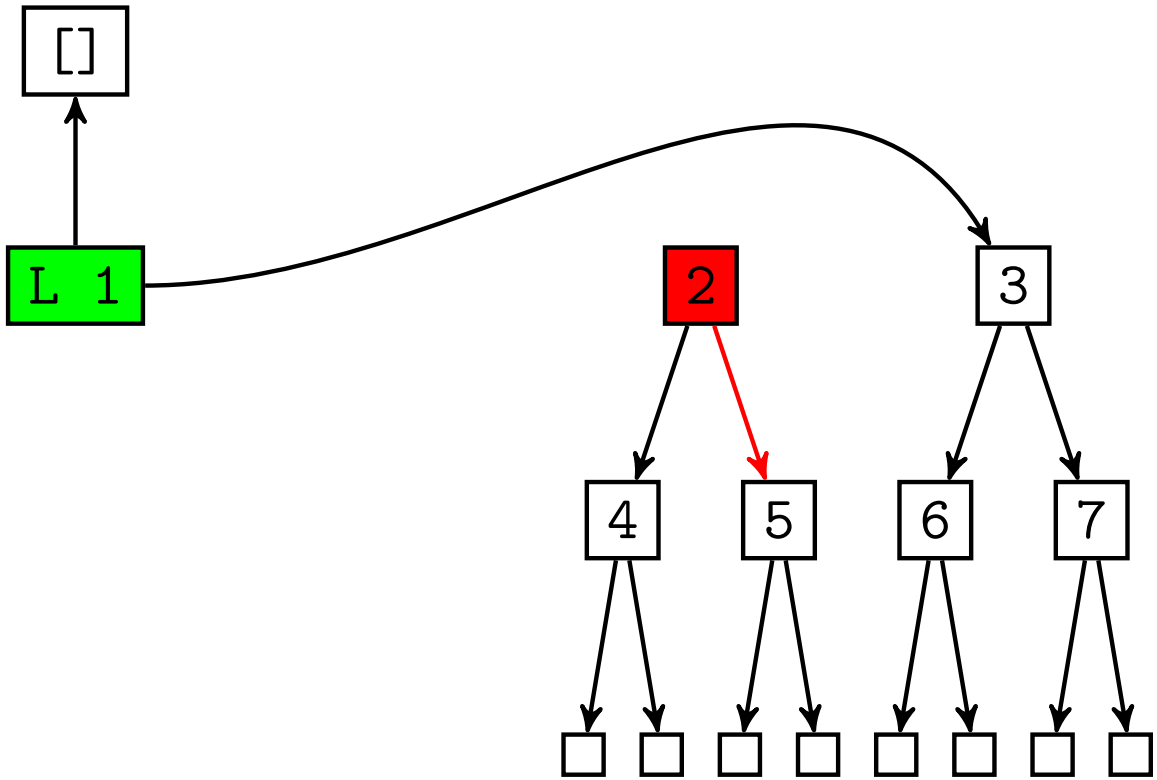

```
goLeft zipper
```

Stromy: příklad



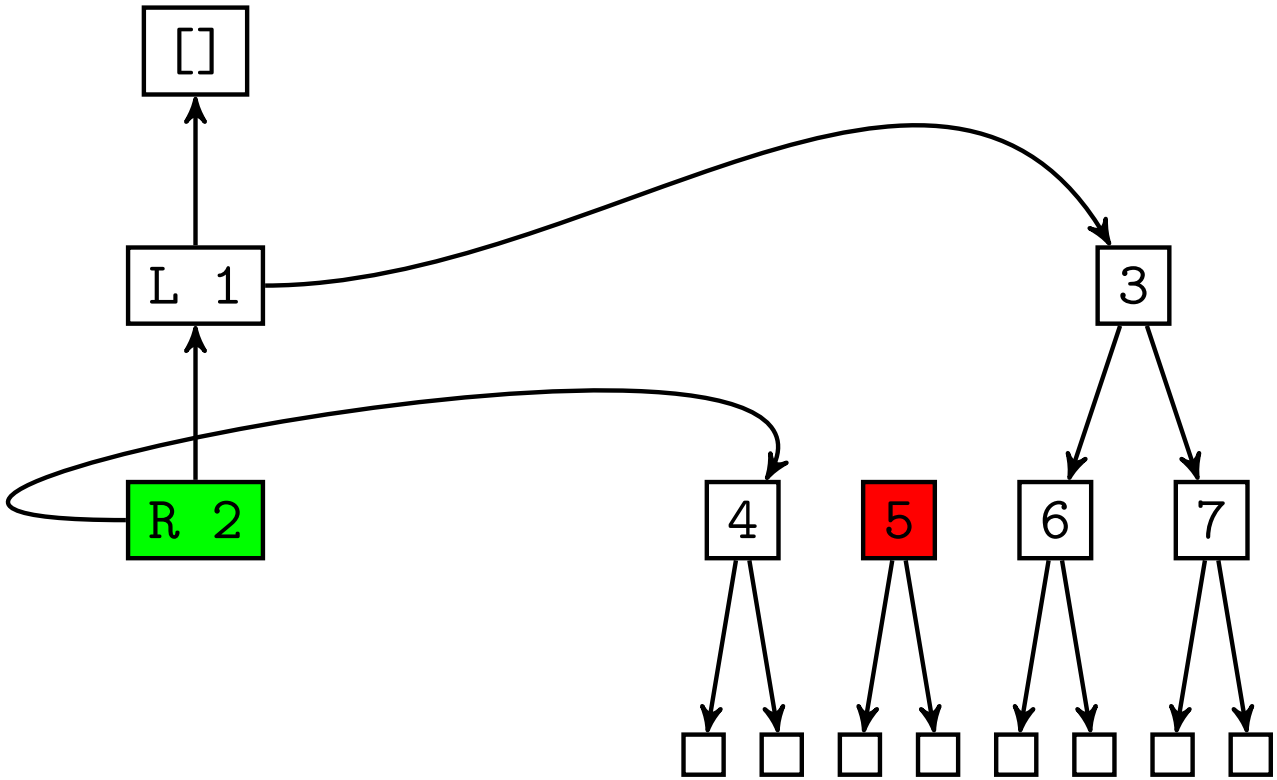

```
zipper TZip [L 1 (N (N E 6 E) 3 (N E 7 E))] (N (N E
4 E) 2 (N E 5 E))
```

Stromy: příklad



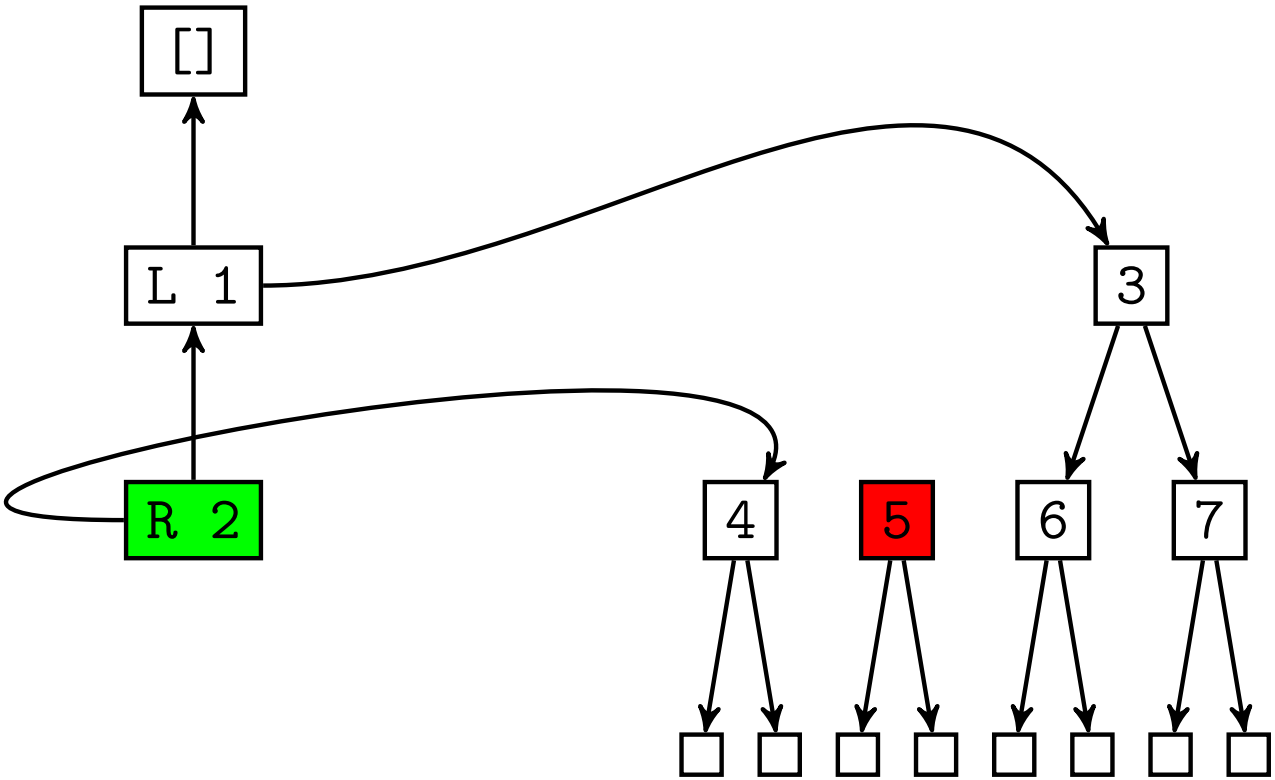

```
goRight zipper
```

Stromy: příklad



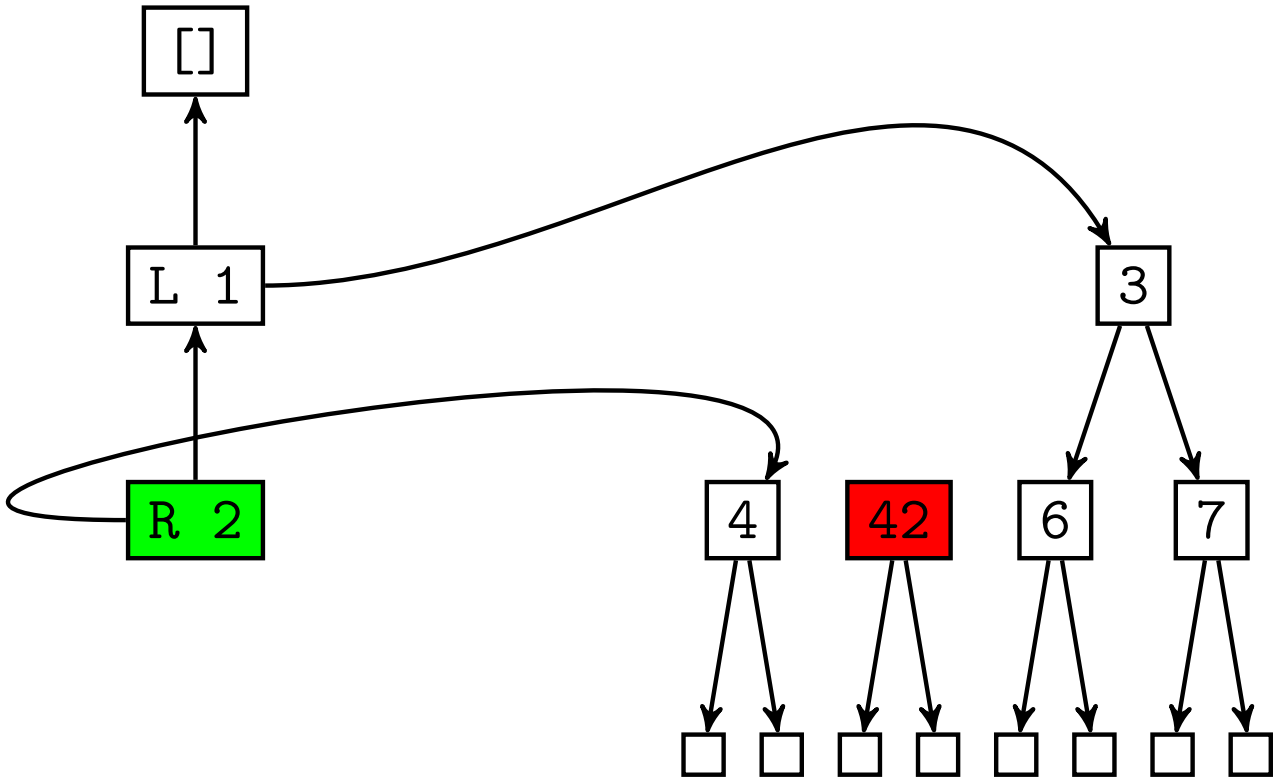

```
zipper TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N E
7 E))] (N E 5 E)
```

Stromy: příklad



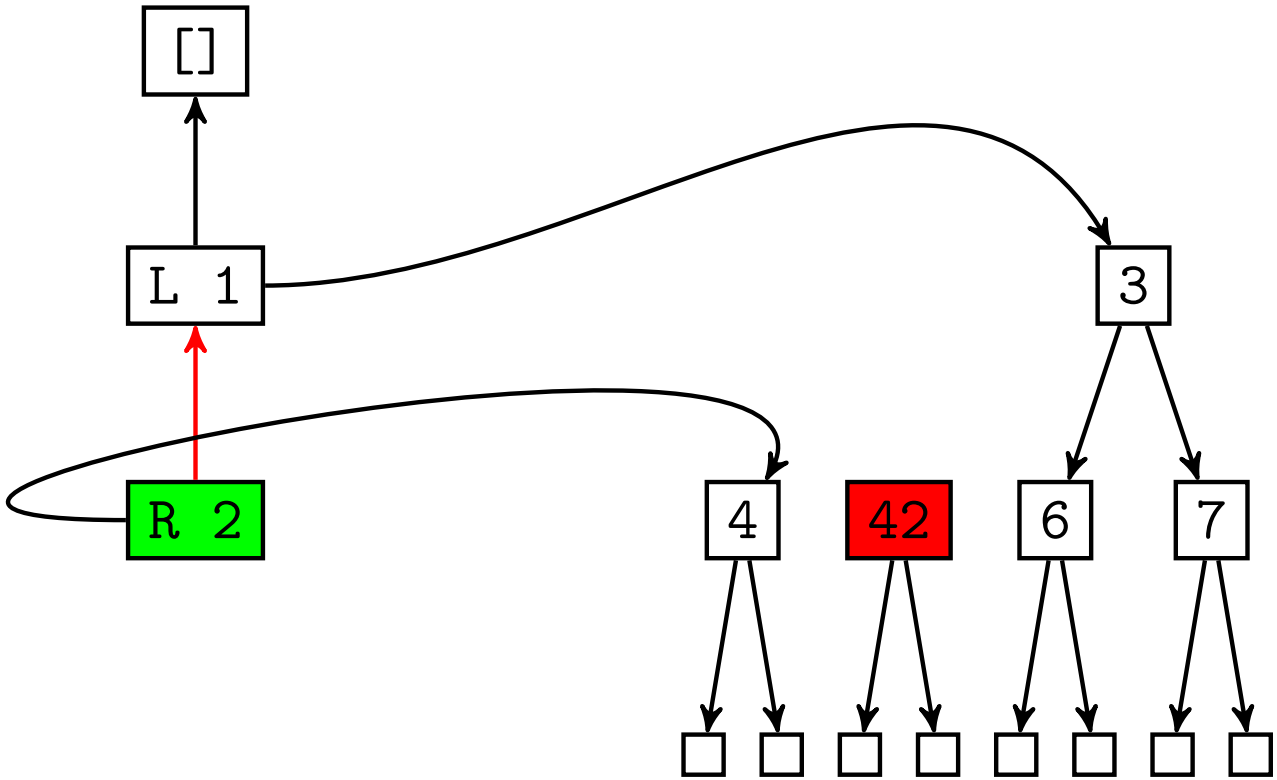

```
modify (+ 37) zipper
```

Stromy: příklad



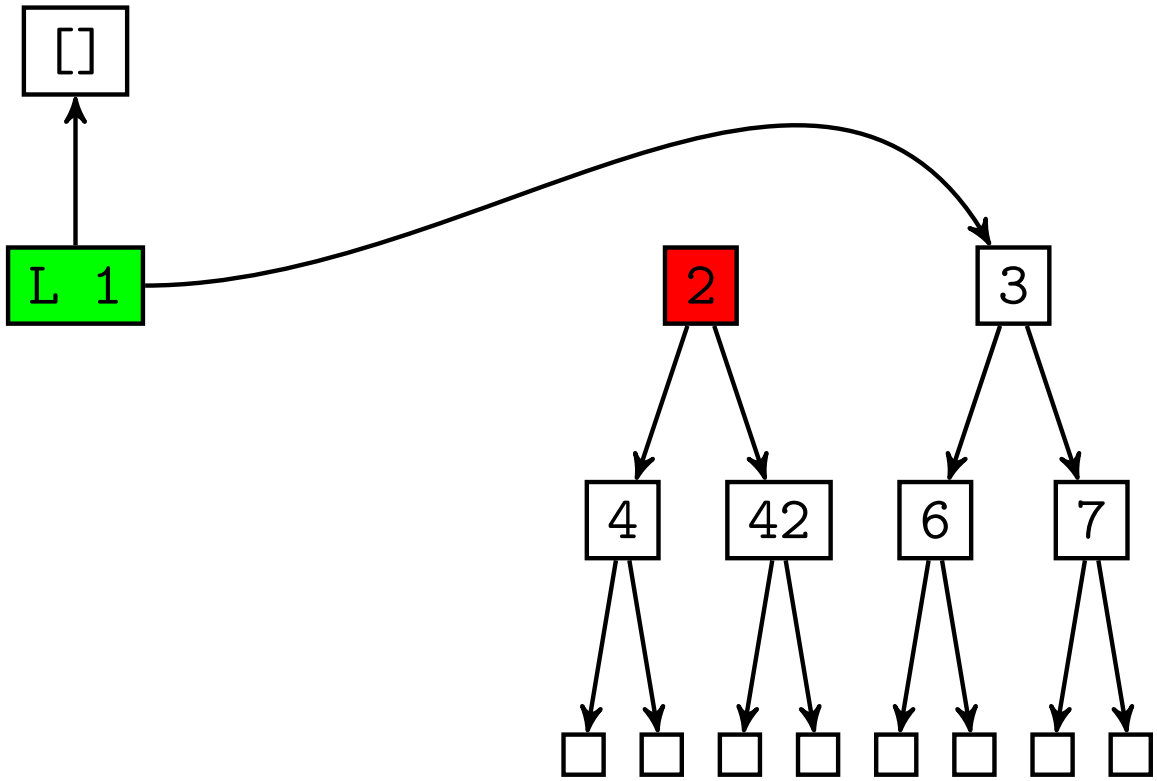

```
zipper = TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N
E 7 E))] (N E 42 E)
```

Stromy: příklad




```
goUp zipper
```

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))
```

Zipper

- obecně zipper pro danou datovou strukturu je datová struktura obohacená o „kontext“, který umožňuje efektivně manipulovat pozici ve struktuře
- pro seznam: `data ListZip a = LZip [a] [a]`
  dvojice seznamů: jeden obsahuje dosud neprojitý seznam, druhý prvky, které již byly projity v opačném pořadí
- pro strom: strom spolu se seznamem kroků zpět ke kořeni – vrchol + levý nebo pravý podstrom
- obdobně pro složitější struktury