Functional Programming (Sheet #6)

Marius Gavrilescu

1. (a)
```
last' (Snoc _ a) = a
```
   (b)
```
head' (Snoc Emptyendlist a) = a
head' (Snoc (Endlist a) _) = head' a
```
   (c)
```
convert Emptyendlist = []
convert (Snoc (Endlist a) b) = convert a ++ b
```
   (d) About $\frac{n(n+1)}{2}$ steps are needed.

```
convert' :: [a] -> Endlist -> [a]
convert' list Emptyendlist = list
convert' list (Snoc (Endlist a) b) = convert' a (b:list)

-- convert = convert' []
```

2. (a)
```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just a) = f a
```
   (b)
```
instance Functor BTree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Fork ltree rtree = Fork (fmap f ltree) (fmap f rtree)
```
   (c)
```
instance Functor RTree where
    fmap f (Node a []) = Node (f a) []
    fmap f (Node a tree) = Node (f a) (fmap f tree)
```

3.
```
foldMaybe :: (a -> b) -> b -> Maybe a -> b
foldMaybe f e Nothing = e
foldMaybe f e Just x = (f x)

foldBTree :: (b -> b) -> (a -> b) -> BTree a -> b
foldBTree f g (Leaf a) = g a
foldBTree f g (Fork (BTree l) (BTree r)) = f (foldBTree f g l) (foldBTree f g r)

foldRTree :: (a -> [b] -> b) -> RTree a -> b
foldRTree f (Node x list) = f a (map (foldRTree f) list)

sumBTree = foldBTree id (+)
sumRTree = foldRTree func where func a list = a + sum list
```

4.
```
findpath (Leaf x) val
    | x == val  = []
    | otherwise = Nothing

findpath (Fork (BTree l) (BTree r))) val
    | a /= Nothing = Left:a
    | b /= Nothing = Right:b
    | otherwise    = Nothing
    where a = findpath l val
          b = findpath r val
```