CS 558 — 26 September 2024

Note Title 2024-09-26

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

This version is quadratic in the overall length.

```
revcat :: [a] -> [a] -> [a]
revcat xs ys = reverse xs ++ ys
```

If we implement revcat, we can then implement reverse in terms of revcat:

```
reverse xs = revcat xs []
```

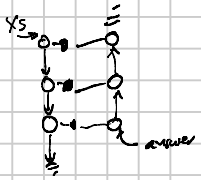
We need a definition of revcat without reference to reverse, e.g. a direct recursive definition:

```
  revcat [] ys
=   { spec. of revcat }
  reverse [] ++ ys
=   { spec. of reverse }
  [] ++ ys
=   { def. of ++ }
  ys
```

```
  revcat (x:xs) ys
=   { spec. of revcat }
  reverse (x:xs) ++ ys
=   { spec. of reverse }
  (reverse xs ++ [x]) ++ ys
=   { associativity of ++ }
  reverse xs ++ ([x] ++ ys)
>   { by def. of ++ }
  reverse xs ++ (x:ys)
=   { by spec. of revcat }
  revcat xs (x:ys)
```

The new definition for reverse:

```
reverse :: [a] -> [a]
reverse xs = revcat xs []
  where
    revcat :: [a] -> [a] -> [a]
    revcat []     ys = ys
    revcat (x:xs) ys = revcat xs (x:ys)
```



Augmented trees with size information

```
data Atree a = Leaf a
             | Fork Int (Atree a) (Atree a)
```

invariant will be that in a `Fork n xt yt`, `n = size xt`

The invariant must be enforced when we build the tree,
so we will only build Fork nodes using a smart fork

```
fork :: Atree a -> Atree a -> Atree a
fork xt yt = Fork (lSize xt) xt yt

lSize :: Atree a -> Int
lSize (Leaf x) = 1
lSize (Fork n xt yt) = n + lSize yt

mkAtree :: [a] -> Maybe (Atree a)
mkAtree xs =
  case xs of
    []  -> Nothing
    [x] -> Just (Leaf x)
    _   -> case mkAtree ys of
             Nothing -> Nothing
             Just yt -> case mkAtree zs of
                          Nothing -> Nothing
                          Just zt -> Just (fork yt zt)
  where
    m = length xs `div` 2
    (ys,zs) = splitAt m xs
```

Exercise:
→ Try with error instead of the Maybe
→ Does it actually work?

Spec.
```
retrieve :: Atree a -> Int -> a
retrieve xt k = (flatten xt) !! k

retrieve (Leaf x) 0 = x
retrieve (Fork m xt yt) k
  | k < m  = retrieve xt k
  | k >= m = retrieve yt (k-m)
```