Lecture 3

appending

```
> [] ++ ys = ys
> (x:xs) ++ ys = x : (xs ++ ys)

-- if we are not careful about our use of append, we might get smth like this
(((a ++ b) ++ c) ++ d)
-- bad for performance, we traverse a multiple times when we don't need to
-- these are "left nested" applications of append

-- api is when fuctionality is given to the code writer in forms of
-- abstractions. a good api would allow all uses of functionality to be
-- "good" no matter how the code writer uses them.

-- we want to make it so that whenever we use multiple appends, it will
-- always be right nested.
-- define the only way you can do appends is by doing right appends
> :t (\xs -> [True] ++ xs)
(\xs -> [True] ++ xs) :: [Bool] -> [Bool]

-- how about we use something called DLists?
> ... :: [a] -> [a]
-- type synonyms allow us to use shortcuts. it is a name, where the name
-- has a one to one correspondence with a type.
> type DList a = [a] -> [a]
>
```

If we were defining a list for other programmers to use instead of regular haskell lists, what would they expect out of it?

- they would want it to be generic
- we have to be able to create a list
    - might want a copyconstructor type thing.

```
> type DList a = [a] -> [a]
> fromList :: [a] -> (DList a)   -- or could be ([a] -> [a])
fromList xs = (\ys -> xs ++ ys)
-- no matter how these lists are used, they will always be
-- right nested appends
toList :: DList a -> [a]
```

```
toList dl = dl []
-- we should provide an equivalent for cons and the empty list in our api
cons :: a -> DList a -> DList a
cons x xs =

empty :: DList a
empty = (\xs -> xs)   -- or (\xs -> [] ++ xs)
```