Lecture Functions

id

```haskell
id :: a -> a
id x = x
```

Takes in one input of any data type, and outputs the input

const

```haskell
const :: a -> b -> a
const x y = x
```

Always returns the first input; `const 5` always returns 5, for example.

sum

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs

Alternative:
sum :: [Int] -> Int
sum = foldr (+) 0
```

length

```haskell
length :: [a] -> Int
length [] = 0
length (x : xs) = 1 + length xs

Alternative:
length :: [a] -> Int
length = foldr (const (1 +)) 0
```

take

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x : xs) = x : take (n - 1) xs
```

drop

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (x : xs) = drop (n - 1) xs
```

succ

```
succ :: Int -> Int
succ x = x + 1
```

map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = (f x) : (map f xs)
```

Properties of map

Map Fusion

```
(map g) . (map f) = map (g . f)
```

(.)

```haskell
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g x = f (g x)
```

filter

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x : xs)
  | p x       = x : (filter p xs)
  | otherwise = filter p xs
```

zip

```haskell
zip :: [a] -> [b] -> [(a, b)]
zip (x : xs) (y : ys) = (x, y) : (zip xs ys)
zip _ _ = []
```

zipWith

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x : xs) (y : ys) = (f x y) : (zipWith f xs ys)
zipWith _ _ _ = []
```

unzip

```haskell
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([], [])
unzip ((x, y) : xys) = (x : xs, y : ys)
where (xs, ys) = unzip xys
```

curry

```haskell
curry :: ((a, b) -> c) -> (a -> b -> c)
```

```
curry f x y = f (x, y)
```

Converts a function that takes in a pair of values into one that takes a value and returns a function (i.e. a function that takes in two values)

uncurry

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (x, y) = f x y
```

The opposite

elem

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem v (x : xs) = (v == x) || (elem v xs)
```

insert

```
insert :: Ord a => a -> [a] -> [a]
insert v [] = [v]
insert v (x : xs)
  | v <= x    = v : x : xs
  | otherwise = x : insert v xs
```

Inserts a value into a sorted list so that it stays sorted.

repeat

```
repeat :: a -> [a] -- Infinite List
repeat x = x : repeat x
```

Better version:

```haskell
repeat :: a -> [a] -- List of size 1
repeat x = xs
  where xs = x : xs
```

(++)

```haskell
(++) :: [a] -> [a] -> [a]
-- [1, 2] ++ [3, 4] = [1, 2, 3, 4]
```

(:)

```haskell
(:) :: a -> [a] -> [a]
-- 1 : [2, 3] = [1, 2, 3]
```

(!!)

```haskell
(!!) :: [a] -> Int -> a
-- Get element of list
```

foldr

Left to Right.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ empty [] = empty
foldr cons empty (x : xs) = cons x (foldr cons empty xs)
```

Not tail recursive.

foldl

Right To Left.

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ empty [] = empty
foldl cons empty (x : xs) = foldl cons empty' xs
  where !empty' = cons empty x
```

Tail recursive.

($)

Called "apply".

```haskell
($) :: (a -> b) -> a -> b
f $ x = f x
```

Don't use!