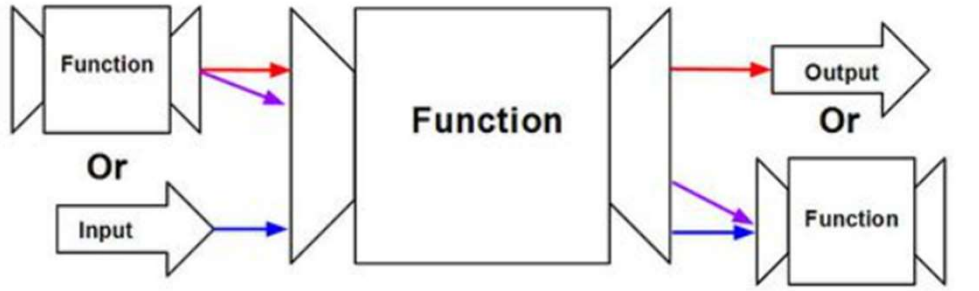
Higher-Order functions

Principles of Programming Languages



Higher-Order functions

A *higher-order function* is technically *any function that takes another function as an argument* or returns a function as result.

- The following are all **higher-order functions**:

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

- These are particularly useful in that they allow us to create new functions on top of the ones we already have, by passing functions as arguments to other functions. Hence the name, **higher-order functions**.

Example

- **even** is a first-order function
- **map** is a higher-order function.

```
Prelude> even 1
False
Prelude> even 2
True
Prelude> map even [1,2,3,4,5]
[False,True,False,True,False]
Prelude> filter even [1,2,3,4,5]
[2,4]
```

Types of Higher-order functions

```
map    :: (a -> b)    -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

- The first argument is a function in both cases.
- They are also polymorphic. This is a powerful combination!

Higher Order Functions

- Functions can take functions as parameters and return functions.
- Consider a function that takes a function and then applies it twice to something.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

- Observe that the first parameter is a function (of type a -> a) and the second is that same a. The function can also be Int -> Int or String -> String or whatever. But then, the second parameter to also has to be of that type.

Higher Order Functions

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

- We just use the parameter f as a function, applying x to it by separating them with a space and then applying the result to f again

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

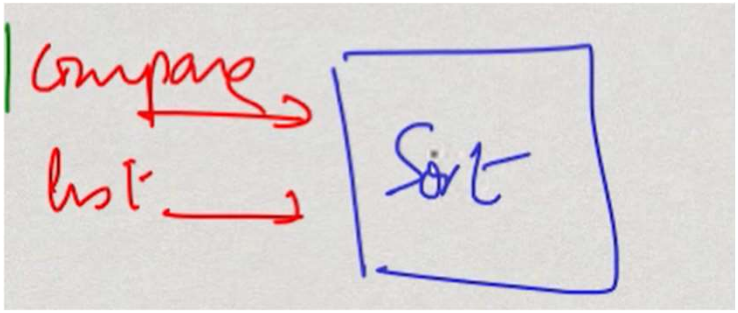
Higher order function

- Can pass functions as argument
- **apply f x = f x**
  - Applies first argument to second argument
- What is the type of apply?
  - A generic function **f** has a type **f :: a -> b**
  - Argument **x** and output must be compatible with f

- apply :: (a -> b) -> a -> b

Higher order functions

- Sorting a list of objects
  - Need to compare pairs of objects.
  - What quantity is used for comparison.
  - Ascending . Descending.
- Pass a comparison function along with the list to the sort function.



Higher order function - Example

- Consider a function zipWith that takes a function and two lists as parameters and then joins the two lists by applying the function between corresponding elements.

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

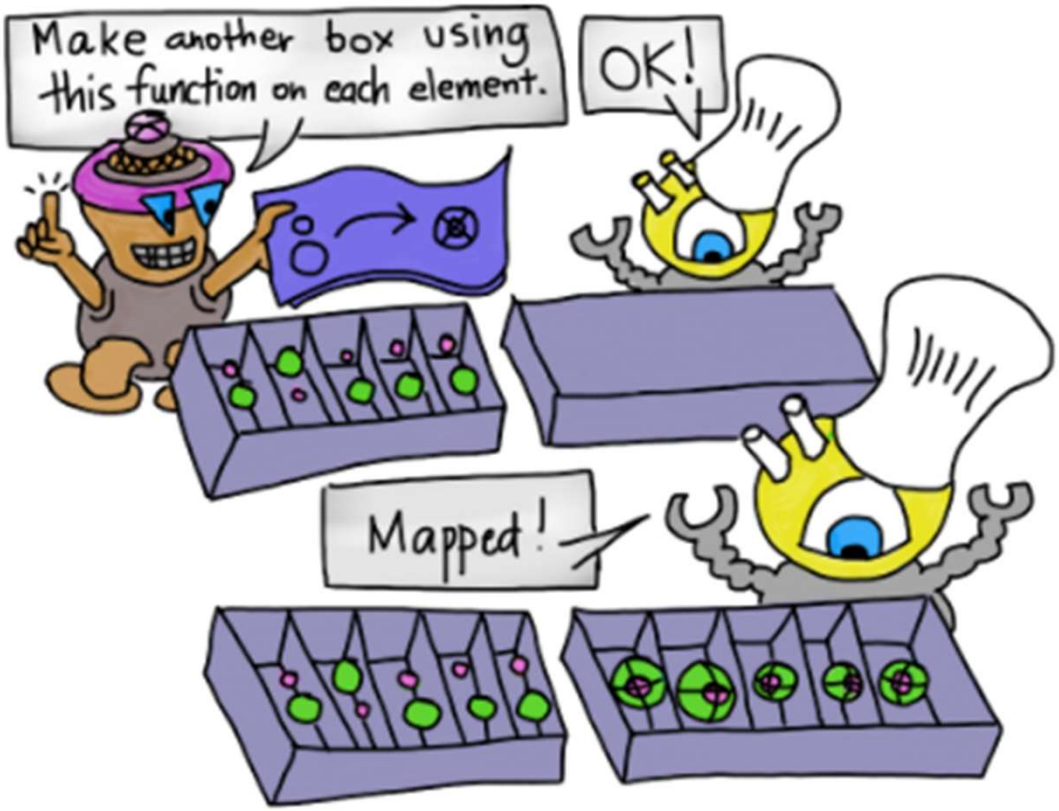
The first parameter is a function that takes two things and produces a third thing. They don't have to be of the same type, but they can. The second and third parameter are lists. The result is also a list.

Higher order function - Example

- Small demonstration of all the different things our zipWith' function can do

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

map function



map function

A map is a higher-order function that requires a list and another function. The *other* function is applied to the list and returns a new list.

- The type of the `map` function then you see:

```
map :: (a -> b) -> [a] -> [b]
```

- The `map` is provided as part of the Haskell's base prelude (i.e. "standard library") and is implemented as:

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

The type signature says that it takes a function that takes an a and returns a b, a list of a's and returns a list of b's.

map function

- The higher-order library function called map a function to every element of a list.

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
```

- Example:-

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]
```

map function

- The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x <- xs]
```

- Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f (x:xs) = f x : map f xs
```

map function

- When it comes to writing maps, Haskell takes a function and a list, then applies that function to every element in the list to produce a new list.

```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
```

- The function (a -> b) is an exact copy transformation from a to b, which means that the mapped array of [b] is going to be exactly the same as [a].

map function

- map is one of those really versatile higher-order functions that can be used in millions of different ways.

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

You've probably noticed that each of these could be achieved with a list comprehension. map (+3) [1,5,3,1,6] is the same as writing [x+3 | x <- [1,5,3,1,6]]. However, using map is much more readable for cases where you only apply some function to the elements of a list

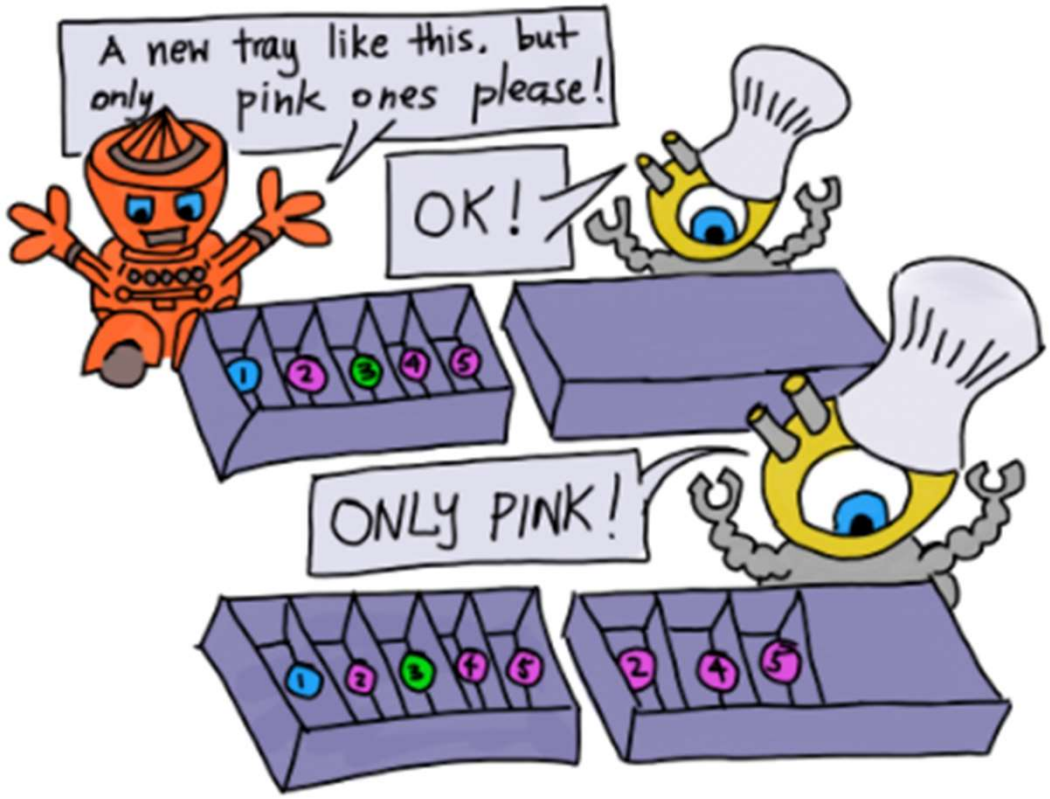
map function

- Using map, we can also do map (∗) [0..], which illustrates how currying works and how (partially applied) functions are actual values that we can pass around to other functions or put into lists.
- map (∗) [0..] produces an infinite list of functions like the one we'd get by writing [(0∗), (1∗), (2∗), (3∗),(4∗),(5∗),...].

```
let listOfFuns = map (*) [0..]
ghci 11> (listOfFuns !! 4) 5
      20
```

Getting the element with the index 4 from our list returns a function that's equivalent to (4∗). And then, we just apply that function to 5. So that's like writing (4∗) 5 or more simply, 4∗5.

filter function



filter : Selecting elements in a list

- We often want to select some of the elements out of a list. For this, we have the **filter** function.
- **filters** in Haskell work by observing all the values it is given and deciding if it qualifies to be a returned value or not.
- **If the parameters of the qualification are not met, the value is not returned**.
- When you filter something, the returned value cannot be reversed.
  - it omits the unqualified values from the original list, meaning that the data is lost once the filter has completed its iteration over the data set.

filter function

- The higher-order library function filter selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

- Example:-

```
ghci> filter even [1..10]
[2,4,6,8,10]
```

filter function

- Filter can be defined using a list comprehension:

```
filter p xs = [x | x <- xs, p x]
```

- Alternatively, it can be defined using recursion:

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

If p x evaluates to True, the element gets included in the new list. If it doesn't, it stays out.

Examples of filter

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
"GAYBALLS"
```

Combining map and filter

- Extract all the vowels in the input and capitalize them.
  - **filter** extracts the vowels, **map** capitalizes them

```
cap_vow :: [Char] -> [Char]
cap_vow l = map touppercase (filter is_vowel l)

is_vowel :: Char -> Char
is_vowel c =  (c=='a') || (c=='e') ||
              (c=='i') || (c=='o') ||
              (c=='u')
```

- Squares of even numbers in a list

```
sqr_even :: [Int] -> [Int]
sqr_even l = map sqr (filter is_even l)
```

Next – Folds in Haskell