

# Class 3: Recursion Patterns

January 30

## some announcements

1. office hours —  
please arrive in first fifteen minutes or email ahead
2. late days —  
please email me before class

map

```
absAll :: [Int] -> [Int]
absAll [] = []
absAll (x : xs) = abs x : absAll xs
```

```
squareAll :: [Int] -> [Int]
squareAll [] = []
squareAll (x : xs) = x * x : squareAll xs
```

```
add3All :: [Int] -> [Int]
add3All [] = []
add3All (x : xs) = x + 3 : add3All xs
```

suspiciously similar examples

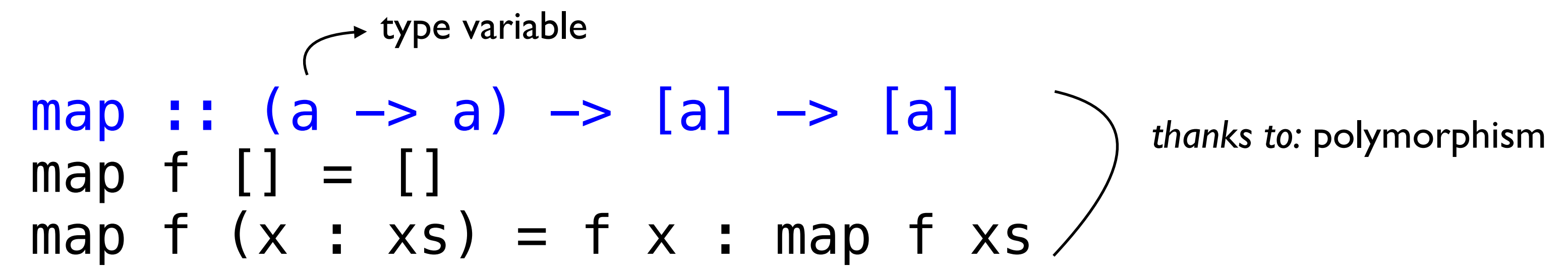
```
map ::                -> [Int] -> [Int]
map [] = []
map (x : xs) = x : map xs
```

first attempt at generalization

*thanks to: first-class functions*

```
map :: (Int -> Int) -> [Int] -> [Int]
map f [] = []
map f (x : xs) = f x : map f xs
```

first attempt at generalization

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

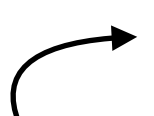
type variable

thanks to: polymorphism

second attempt at generalization


```
absAll :: [Int] -> [Int]
absAll xs = map abs xs
```

```
squareAll :: [Int] -> [Int]
```

 anonymous function

```
squareAll xs = map (\x -> x * x) xs
```

```
add3All :: [Int] -> [Int]
add3All xs = add3All (+ 3) xs
```

 operator section



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

```
roundAll :: [Double] -> [Int]
roundAll xs = map round xs
```

Respond at **PollEv.com/jessicashi159**

Text **JESSICASHI159** to **37607** once to join, then text your message

## What is the type of `length`?

```
length :: ???  
length [] = 0  
length (_ : xs) = 1 + length xs
```

Respond at **PollEv.com/jessicashi159**

Text **JESSICASHI159** to **37607** once to join, then text your message

## What is the type of `length`?

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (_ : xs) = 1 + length xs
```

🌐 Respond at **PollEv.com/jessicashi159**

📱 Text **JESSICASHI159** to **37607** once to join, then text your message

## What is the type of `lengthAll`?

```
lengthAll :: ???
```

```
lengthAll xs = map length xs
```

🌐 Respond at **PollEv.com/jessicashi159**

📱 Text **JESSICASHI159** to **37607** once to join, then text your message

## What is the type of `lengthAll`?

```
lengthAll :: [[a]] -> [Int]  
lengthAll xs = map length xs
```

**filter**

```
upperOnly :: [Char] -> [Char]
upperOnly [] = []
upperOnly (x : xs)
  | isUpper x = x : upperOnly xs
  | otherwise = upperOnly xs
```

```
positiveOnly :: [Int] -> [Int]
positiveOnly [] = []
positiveOnly (x : xs)
  | x > 0 = x : positiveOnly xs
  | otherwise = positiveOnly xs
```

suspiciously similar examples

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

generalization



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

generalization

```
upperOnly :: [Char] -> [Char]  
upperOnly xs = filter isUpper xs
```

```
positiveOnly :: [Int] -> [Int]  
positiveOnly xs = filter (> 0) xs
```

*(people exercise)*

fold

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

```
product :: [Int] -> Int
product [] = 1
product (x : xs) = x * product xs
```

suspiciously similar examples

base case for empty list

```
sum :: [Int] -> Int
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

```
product :: [Int] -> Int
```

```
product [] = 1
```

```
product (x : xs) = x * product xs
```

suspiciously similar examples

combine first element with...

```
sum :: [Int] -> Int
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

```
product :: [Int] -> Int
```

```
product [] = 1
```

```
product (x : xs) = x * product xs
```

suspiciously similar examples

result for rest of list

```
sum :: [Int] -> Int
```

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

```
product :: [Int] -> Int
```

```
product [] = 1
```

```
product (x : xs) = x * product xs
```

suspiciously similar examples



```
fold ::  
fold [] =  
fold (x : xs) =
```

```
[a] -> a
```

first attempt at generalization

```
fold ::  
fold  z [] = z  
fold  z (x : xs) =
```

starting value  
a -> [a] -> a

first attempt at generalization

current element

starting value

```
fold :: (a -> a -> a) -> a -> [a] -> a
fold f z [] = z
fold f z (x : xs) = f x (
```

first attempt at generalization

current element      accumulated value      starting value

`fold :: (a -> a -> a) -> a -> [a] -> a`

`fold f z [] = z`

`fold f z (x : xs) = f x (fold f z xs)`

first attempt at generalization

current element      accumulated value      starting value

$\swarrow$                        $\uparrow$                        $\swarrow$

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

`fold f z [] = z`

`fold f z (x : xs) = f x (fold f z xs)`

second attempt at generalization

current element      accumulated value      starting value

$\swarrow$                        $\uparrow$                        $\nearrow$

`fold :: (a ->      ->      ) ->      -> [a] -> b`  
`fold f z [] = z`  
`fold f z (x : xs) = f x (fold f z xs)`

second attempt at generalization

current element      accumulated value      starting value

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

`fold f z [] = z`

`fold f z (x : xs) = f x (fold f z xs)`

second attempt at generalization

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

```
prod :: [Int] -> Int  
prod xs = fold (*) 1 xs
```



```
length :: [a] -> Int  
length xs = fold
```

xs

another example

```
length :: [a] -> Int  
length xs = fold
```

0 xs

another example

```
length :: [a] -> Int
length xs = fold (\x l ->      ) 0 xs
```

another example

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

another example

```
length :: [a] -> Int
length xs = fold (\_ l -> 1 + l) 0 xs
```

another example

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)
```

```

foldr (-) 0 [3, 2, 1]
= foldr (-) 0 (3 : (2 : (1 : [])))
=                (3 - (2 - (1 - 0)))

```

```

foldl (-) 0 [3, 2, 1]
= foldl (-) 0 (3 : (2 : (1 : [])))
=                (((0 - 3) - 2) - 1)

```

foldr vs. foldl

*(reimplement exercise)*



Hoogle

*(map example)*

🌐 Respond at **PollEv.com/jessicashi159**

📱 Text **JESSICASHI159** to **37607** once to join, then text your message

**Find a function `foo` that gets the first  $n$  elements.**

`foo 2 [1, 2, 3] = [1, 2]`

🌐 Respond at **PollEv.com/jessicashi159**

📱 Text **JESSICASHI159** to **37607** once to join, then text your message

**Find a function `foo` that checks whether some element is in the list.**

```
foo 3 [1, 2, 3] = True
```

🌐 Respond at **PollEv.com/jessicashi159**

📱 Text **JESSICASHI159** to **37607** once to join, then text your message

**Find a function `foo` that combines the elements of two lists using some function.**

`foo (+) [1, 2] [3, 4] = [4, 6]`

🌐 Respond at **PollEv.com/jessicashi159**

📱 Text **JESSICASHI159** to **37607** once to join, then text your message

**Find a function `foo` that splits a string into words.**

```
foo "these are words" = ["these", "are", "words"]
```