

Class 3: Recursion Patterns

February 6

```
data Exp
  = Num Nat
  | Add Exp Exp
  | Sub Exp Exp
  | Mul Exp Exp
```

questions from homework

`"(3 + (4 * 5)) - 6"`

program text

parses to

`(Sub (Add 3 (Mul 4 5)) 6)`

abstract representation of
the programming language

evaluates to

17

result value

questions from homework

program text



parses to

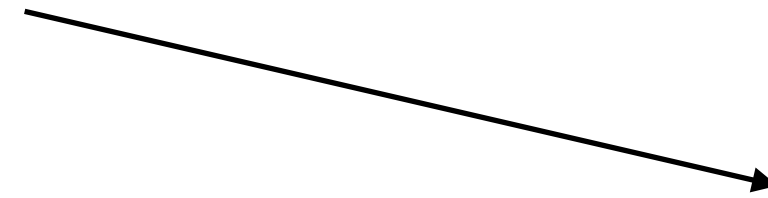
abstract representation of
the programming language



evaluates to

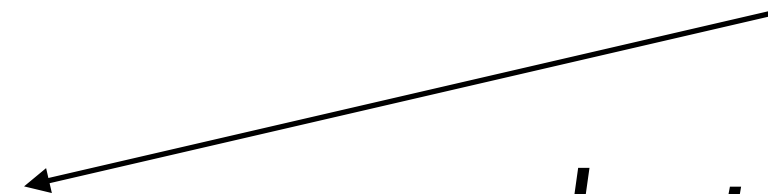
result value

compiles to



lower-level representation of
the same language

evaluates to



questions from homework

(note: this week's autograder)
(note: looking forward)

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

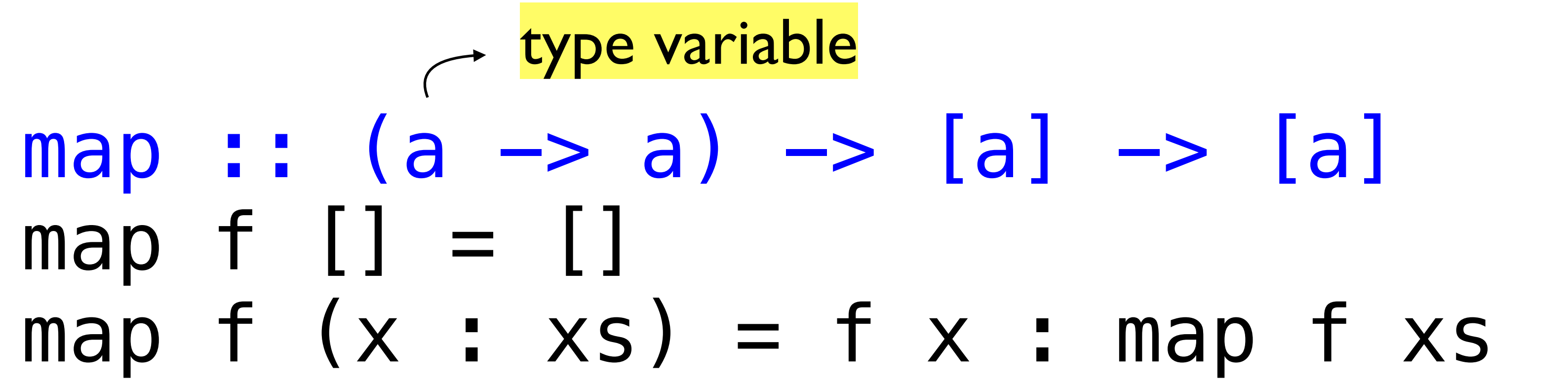
thanks to: first-class functions

`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

thanks to: polymorphism

second attempt at generalization

```
absAll :: [Int] -> [Int]
```

```
absAll xs = map abs xs
```

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

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

anonymous function

```
add3All :: [Int] -> [Int]
```

```
add3All xs = map (\x -> x + 3) xs
```

examples revisited

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

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

final version

length :: ???

length [] = 0

length (_ : xs) = 1 + length xs

length :: [a] -> Int

length [] = 0

length (_ : xs) = 1 + length xs

lengthAll :: ???

lengthAll xs = map length xs

```
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 (\x -> x > 0) xs
```

examples revisited

(exercise: young names)

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

sum :: [Int] -> Int

sum [] = 0

base case

sum (x : xs) = x + sum xs

product :: [Int] -> Int

product [] = 1

product (x : xs) = x * product xs

suspiciously similar examples

`sum :: [Int] -> Int`

`sum [] = 0`

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

combine first
element with...

`product :: [Int] -> Int`

`product [] = 1`

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

suspiciously similar examples

sum :: [Int] -> Int

sum [] = 0

sum (x : xs) = x + sum xs result for rest of list

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) =`

`a -> [a] -> a`
starting value

first attempt at generalization

current element

$\text{fold} :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a$

$\text{fold } f \ z \ [] = z$

$\text{fold } f \ z \ (x : xs) = f \ x \ (\quad)$

first attempt at generalization

current element accumulated value

$\text{fold} :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a$

$\text{fold } f \ z \ [] = z$

$\text{fold } f \ z \ (x : xs) = f \ x \ (\text{fold } f \ z \ xs)$

first attempt at generalization

current element

accumulated value

new value

$\text{fold} :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a$

$\text{fold } f \ z \ [] = z$

$\text{fold } f \ z \ (x : xs) = f \ x \ (\text{fold } f \ z \ xs)$

first attempt at generalization


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

second attempt at generalization

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

second attempt at generalization

```
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
```

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

examples revisited

idiom:

Functions can be used without mentioning their arguments.

technique:
Use function names directly.

```
map (\x -> abs x) xs
```

vs.

```
map abs xs
```

technique:

Use function names directly — even for operators.

```
fold (\x y -> x + y) 0 xs
```

vs.

```
fold (+) 0 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 vs. foldl

```

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

(exercise: reimplement map, filter)

Hoogle

(example: searching for map)

🌐 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]`

addendum: polymorphic data types

```
data FailableDouble  
  = Failure  
  | OK Double
```

```
data WeatherResult  
  = Invalid  
  | Valid Weather
```

suspiciously similar examples

type constructor

type variable

```
data Maybe a
  = Nothing
  | Just a
```

```
graph TD
    TC[type constructor] --> M[Maybe]
    TV[type variable] --> a[a]
```

maybe

```
safeDiv :: Double -> Double -> Maybe Double  
safeDiv _ 0 = Nothing  
safeDiv m n = Just (m / n)
```

maybe

```
data List a
  = Nil
  | Cons a (List a)
```

lists


```
safeHead :: [a] -> Maybe a  
safeHead (x : _) = Just x  
safeHead [] = Nothing
```

polymorphic functions on polymorphic data types