# Class 5: Review*

February 20

# readability

Q: is writing code like this really more *readable*?

question from homework

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

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

a lateral move

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

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

remember this?

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

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

remember this?

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

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

remember this?

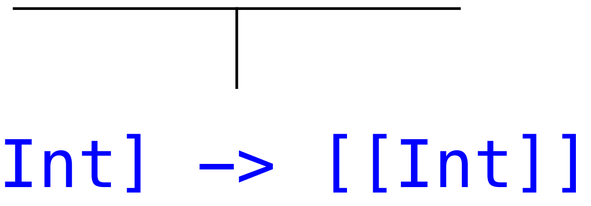
```
toRevDigits :: Int -> [Int]
sum :: [Int] -> Int

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

remember this?

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

sumDigits :: [Int] -> Int
sumDigits = sum . map sum . map toRevDigits
```

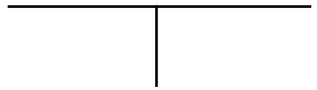
now look at this!

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

sumDigits :: [Int] -> Int
sumDigits = sum . map sum . map toRevDigits
```




now look at this!

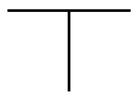
```
toRevDigits :: Int -> [Int]
sum :: [Int] -> Int

sumDigits :: [Int] -> Int
sumDigits = sum . map sum . map toRevDigits
```



`[[Int]] -> [Int]`

now look at this!

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

sumDigits :: [Int] -> Int
sumDigits = sum . map sum . map toRevDigits
```



```
[Int] -> Int
```

now look at this!

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

sumDigits :: [Int] -> Int
sumDigits = sum . map sum . map toRevDigits
```

now look at this!

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

sumDigits :: [Int] -> Int
sumDigits = sum . map (sum . toRevDigits)
```

but what about this?

A: concision is not the goal, though it is a side effect.
the goal is to reason about problems at a higher level,

so that our code can focus on communicating
*what* it does instead of *how* it does it.

question from homework

# more (and less) pattern matching

Q: why does pattern matching work?

question from homework

```
exec :: [Instr] -> Stack -> Stack
exec (i : is) st = exec is (exec1 i st)
exec is       st = st
```

stylistic recommendation: avoid “catch all” cases that only match one case

```
exec :: [Instr] -> Stack -> Stack
exec (i : is) st = exec is (exec1 i st)
exec []       st = st
```

stylistic recommendation: avoid “catch all” cases that only match one case

```
eval :: Exp -> Nat
…
eval (Branch e1 e2 e3) = case eval e1 of
                           Z -> eval e2
                           S _ -> eval e3
…
```

case expressions

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

case expressions

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

case expressions

```
remove7 :: [[Int]] -> [[Int]]
remove7 [] = []
remove7 (xs : xss) = map (…) (xs : xss)
```

stylistic recommendation: avoid unnecessary pattern matching

```
remove7 :: [[Int]] -> [[Int]]
remove7 xss = map (…) xss
```

stylistic recommendation: avoid unnecessary pattern matching

```
same :: [Int] -> Bool
same [] = True
same (x : xs) = all (== x) xs
```

cf. useful pattern matching

follow the types!

```
data WeatherForecast
  = Forecast Weather
  | Unknown


data Maybe a
  = Nothing
  | Just a
```

refactoring example

*(example: refactoring to use Maybe)*

```
f :: a -> a -> a
```

revisiting polymorphism

```
f :: a -> a -> a
f a1 a2 = a1 && a2
```

```haskell
f :: a -> a -> a
f a1 a2 = case (typeOf a1) of
  Int -> a1 + a2
  Bool -> a1 && a2
  _ -> a1
```

```
f :: a -> a -> a
```

*parametric* polymorphism

```
f :: a -> a -> a
f a1 a2 = a1

f :: a -> a -> a
f a1 a2 = a2
```

*parametric* polymorphism

```
f :: a -> a
```

can we define this function?

```
f :: a -> b
```

can we define this function?

```
f :: (a -> a) -> a
```

can we define this function?

```
f :: (a -> a) -> a -> a
```

can we define this function?

```
f :: [a] -> a
```

can we define this function?

```
f :: (b -> c) -> (a -> b) -> (a -> c)
```

can we define this function?

*(exercise: const and flip)*

your questions

Q: can {functional programming,
an expressive type system}
improve other languages?

question from homework

Q: what is Haskell's test framework like?

A: so far, we've been using HUnit for writing unit tests.
eventually, we'll see QuickCheck for writing *property-based tests*.

question from homework

Q: what's this?

```
concat [] ~?= ([] :: [()])
```

A: ( ) is the *unit type*.

usually, Haskell can infer the type of the elements in the list.

here, it can't because the list is empty. so we provide one!

question from homework

Q: what's this?

```
sum :: (Foldable t, Num a) => t a -> a
```

A: type class constraints!

we'll learn about these right after spring break :)

question from homework