

---

csci54 – discrete math & functional programming  
pattern matching, guards, where bindings

---

# this week

---

- ▶ week03-group
  - ▶ please select pages for each question
- ▶ week03-ps-coding
- ▶ reminders
  - ▶ assignment regrades
  - ▶ missing lectures



## Last time - types

---

- ▶ specifying the type of a function:

```
name :: (typeClass var1, typeClass var1, typeClass var2, ...) =>  
    var1 -> var2 -> returnVal
```



# Practice

---

- ▶ What are the types of the following functions?

```
f1 'a' _ = []  
f1 x y = x:y
```

```
f2 (x:y:z:w:l) = w  
f2 _ = 0
```

- ▶ Discussion:
  - ▶ use of wildcard character \_
  - ▶ what does (x:y:z:w:l) match to?
  - ▶ Are both these definitions exhaustive?
- ▶ What do the functions do?



## Last time – pattern matching

---

- ▶ pattern matching:

```
maxList :: [Integer] -> Integer
maxList [] = error "empty list"
maxList [x] = x
maxList (x:xs) = max x (maxList xs)
```



## More pattern matching

---

- ▶ You can pattern match against multiple lists!
- ▶ Consider this function:

```
equal :: (Eq a) => [a] -> [a] -> Bool
equal [] [] = True
equal _ [] = False
equal [] _ = False
equal (x:xs) (y:ys) =
    if x == y
    then equal xs ys
    else False
```



# Pattern matching

---

- ▶ What breaks if you don't include (only) the 2<sup>nd</sup> pattern?
  - ▶ \*\*\* Exception: Non-exhaustive patterns in function equal'
- ▶ What breaks if you don't include (only) the 1<sup>st</sup> pattern?
  - ▶ will always return False

```
equal' :: (Eq a) => [a] -> [a] -> Bool
equal' [] [] = True
equal' _ [] = False
equal' [] _ = False
equal' (x:xs) (y:ys) =
    if x == y
    then equal' xs ys
    else False
```



## One more practice question

---

- ▶ Consider a function `everyOther` that takes a list and returns a new list consisting of every other element in the original list starting with the first element. As an example, `everyOther [1,5,2,4,-1]` should return `[1,2,-1]`
- ▶ What is the type of `everyOther`?
- ▶ How would you implement `everyOther` using pattern matching?



---

---



## case

---

- ▶ We can also pattern-match within the body of a function:

```
last' xs =  
  case xs of  
    [] -> error "empty list"  
    [x] -> x  
    x:xs -> last' xs
```

- ▶ This can be useful if you need to e.g. make a choice based on the return value of your recursive case
- 
- 

# Guards

---

- ▶ pattern-matching lets you specify cases based on values
- ▶ guards let you specify cases based on expressions
- ▶ can combine the two

```
equal :: (Eq a) => [a] -> [a] -> Bool
equal [] [] = True
equal _ [] = False
equal [] _ = False
equal (x:xs) (y:ys) =
    if x == y
    then equal xs ys
    else False
```

```
equal' :: (Eq a) => [a] -> [a] -> Bool
equal' [] [] = True
equal' _ [] = False
equal' [] _ = False
equal' (x:xs) (y:ys)
    | x == y = equal' xs ys
    | otherwise = False
```

# Where bindings

---

- ▶ Gives you the ability to name intermediate values

```
equal' :: (Eq a) => [a] -> [a] -> Bool
equal' [] [] = True
equal' _ [] = False
equal' [] _ = False
equal' (x:xs) (y:ys)
  | x == y = equal' xs ys
  | otherwise = False
```

```
equal' :: (Eq a) => [a] -> [a] -> Bool
equal' [] [] = True
equal' _ [] = False
equal' [] _ = False
equal' (x:xs) (y:ys)
  | x == y = rest
  | otherwise = False
  where rest = equal' xs ys
```

- ▶ Scope: where bindings are visible to entire function



# Let bindings

---

- ▶ Similar to `where`
- ▶ scope is more localized
  - ▶ does not bind across guards
- ▶ are expressions themselves
  - ▶ syntax is "let <bindings> in <expression>"

```
ghci> 4 * (let a = 9 in a + 1) + 2
```



# Practice

---

- ▶ What does the following function do?

```
import Data.Char
```

importing a module in Haskell  
this one gives us functions including toLower

```
mystery x y
  | aL > 'm' && bL > 'm' = "group 4"
  | aL > 'm' && bL <= 'm' = "group 3"
  | aL <= 'm' && bL > 'm' = "group 2"
  | otherwise = "group 1"
where (a:_) = x
      (b:_) = y
      aL = toLower a
      bL = toLower b
```

- ▶ Code is a little repetitive---how could it be simplified?
-

- 
- Bonus section if we're doing OK on time



# Fallible Functions

---

- ▶ We see in functions like `maxInt` that some cases crash the program
- ▶ These “partial functions” can be tricky to work with
- ▶ What could we do in Python or Java to take the “maximum” of an empty list?
  - ...



# Maybe type

---

- ▶ Let's introduce a new type:

```
data Maybe a =  
    Nothing  
  | Just a  
  deriving (Show, Eq)
```

- ▶ Maybe (aka Option) is common nowadays in C++, Java, TypeScript, Rust, & others
- ▶ We encode “either something or null” into the *type*, rather than as a language feature like undefined or None
- ▶ Then we can just write regular functions on it:

```
orElse :: Maybe a -> a -> a  
orElse (Just a) _ = a  
orElse Nothing b = b
```

# Maybe

---

```
data Maybe a =  
    Nothing  
  | Just a  
  deriving (Show, Eq)
```

- ▶ What's the issue with the code below?
- ▶ `maxInt :: [Integer] -> Maybe Integer`
- ▶ `maxInt [] = Nothing`
- ▶ `maxInt [x] = Just x`
- ▶ `maxInt (x:xs) = max x (maxInt xs)`



# Maybe

---

```
data Maybe a =  
    Nothing  
  | Just a  
  deriving (Show, Eq)
```

- ▶ Will this do the trick?
- ▶ `maxInt :: [Integer] -> Maybe Integer`
- ▶ `maxInt [] = Nothing`
- ▶ `maxInt [x] = Just x`
- ▶ `maxInt (x:xs) = max x (maxInt xs)`      ← a Maybe Integer, not an Integer!
- ▶ `maxInt (x:xs) = max x ((maxInt xs) `orElse` x)`



# Maybe

---

```
data Maybe a =  
    Nothing  
  | Just a  
  deriving (Show, Eq)
```

- ▶ `maxInt :: [Integer] -> Maybe Integer`
- ▶ `maxInt [] = Nothing`
- ▶ `maxInt [x] = Just x`
- ▶ `maxInt (x:xs) = max x ((maxInt xs) `orElse` x)`  
^^^ an Integer, not an Option Integer!
- ▶ `maxInt (x:xs) = Just (max x ((maxInt xs) `orElse` x))`



# Maybe

---

- ▶ `maxInt :: [Integer] -> Maybe Integer`
- ▶ `maxInt [] = Nothing`
- ▶ `maxInt [x] = Just x`
- ▶ `maxInt (x:xs) = Just (max x ((maxInt xs) `orElse` x))`
- ▶ Does this look too complicated?
- ▶ There are ways to make it simpler---e.g. using `map` or `fold` which we'll see next week: `map (max x) (maxInt xs)`
- ▶ Equivalent Python code is actually longer, especially the recursive version
  - AND it's more error-prone
  - In Haskell, if we say we have an Integer, then we definitely have an Integer---not null, not ever.



# Fallible Functions

---

- ▶ Error handling is a big topic.
- ▶ Not something-or-nothing; good-thing-or-bad-thing
- ▶ Haskell has `Either` (`Left a | Right b`), which you can use to return more informative errors (e.g., a file-reading function might return `Either String FileReadError`)
- ▶ Our pattern matching abilities make dealing with optional values straightforward (if verbose)
- ▶ Higher-order functions, which we'll see next week, are even more powerful and concise

