csci54 – discrete math & functional programming
types and pattern matching

this week

- week02-group
  - work on during small group meeting this week tomorrow and the day after
  - due 10pm on Thursday
- week02-ps
  - due 10pm on Sunday

Last time

- Write a function oddList' where oddList' evaluates to a list of odd integers from 1 up to, but possibly not including, n. If n < 1 the function should return an empty list

```
oddList' n =
  if n <= 0
  then []
  else if (n `mod` 2) == 1
       then oddList' (n-1) ++ [n]
       else oddList' (n-1)
```

Types

- If turning in for assignment, would be expected to include a line above it, which might read as follows.

```
oddList' :: Integer -> [Integer]
oddList' n =
  if n <= 0
  then []
  else if (n `mod` 2) == 1
       then oddList' (n-1) ++ [n]
       else oddList' (n-1)
```

- Terms: types, type variables, type classes

Types

- The Haskell type system is fantastic
  - It infers as much as possible and won't let you execute code that doesn't type-check.
  - This can make it very, very frustrating
- common Haskell types
  - Int, Integer
  - Float, Rational, Double
  - Bool
  - Char, String

```
ghci> :t 'a'
ghci> :t "A"
ghci> :t 4==5
```

- what about functions?

```
ghci> check x = (x == True)
ghci> :t check
```

Types – for functions

```
ghci> check x = (x == True)
ghci> :t check
check :: Bool -> Bool
```

```
oddList' :: Integer -> [Integer]
```

- what if a function takes multiple parameters?

```
pow n k =
  if k == 0
  then 1
  else n * (pow2 n (k-1))
```

```
pow :: Integer -> Integer -> Integer
```

```
ghci> :t head
```

Type variables

- In some cases you have a function that could take any type
  - declare the type with a type variable

```
ghci> :t head
head :: [a] -> a
```

- What if you have a function that could take some types, but not all types?

Type classes

- A type class is an interface that defines some behavior
- A type that is an instance of a given type class must support that behavior

```
oddList :: Integral a => a -> [a]
```

- Common type classes
  - Num, Floating, Integral
    - fromIntegral function for converting Integral to more general Num
  - Eq, Ord
  - Enum
  - Show, Read

```
ghci> :t 2
```

Putting it all together

- Basic format:

```
name :: (typeClass typeVar, typeClass typeVar, ...) =>

  var1 -> var2 -> returnVal
```

- What are the types of these functions?
  - That is, what would Haskell infer the types were if we didn't specify explicitly?

```
addTriplet (x, y, z) = x + y + z
addTriplet' x y z = x + y + z
```

a function pythagoras that takes a tuple of integers (a, b, c) and returns True if and only if $a^2 + b^2 = c^2$

Functions and Pattern matching

- Write a function pow that takes two parameters n and k and returns n to the kth power. (assume that k is guaranteed to be a non-negative integer)

```
pow :: Integer -> Integer -> Integer
pow n k =
  if k == 0
  then 1
  else n * (pow n (k-1))
```

- Could also be written as follows

```
pow :: Integer -> Integer -> Integer
pow _ 0 = 1
pow n k = n * (pow n (k-1))
```

Pattern matching

- Idea is to specify patterns for data to match. If matches, then deconstruct the data according to the pattern

- You can pattern match on any data type: numbers, characters, lists, tuples, etc.

- When defining functions, you can define separate function bodies for different patterns.

```
isSeven :: (Integral a) => a -> String
isSeven 7 = "You're right!"
isSeven x = "Sorry!"
```

  - checks the patterns from top to bottom

Pattern matching with lists

- Write a function that takes a list of integers and returns the largest integer in that list:

```
maxInt :: [Integer] -> Integer
maxInt [x] = x
maxInt (x:xs) = max x (maxInt xs)
```

  - [x] matches list with exactly one element
  - (x:xs) matches list with **at least** one element (x matches the first element and xs matches the rest of the list)
  - What happens if you give `maxInt` an empty list?

```
*** Exception: week02-lec03-code.hs:(19,1)-(20,33):
Non-exhaustive patterns in function maxInt
```

Pattern matching with lists

- Write a function that takes a list of integers and returns the largest integer in that list:

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

  - [] matches empty list
  - [x] matches list with exactly one element
  - (x:xs) matches list with **at least** one element (x matches the first element and xs matches the rest of the list)
  - Does it still work if you don't include the 2nd pattern?
  - Does it still work if you reverse the order of the 3 patterns?

Practice problems

- Use pattern matching to write a function `last'` that returns the last element of a list (give an error if the list is empty)

- Use pattern matching to write a function `nextToLast` that returns the second-to-last element of a list