COMP105 Lecture 21

Parameterized Custom Types

Recap: Custom Types

```haskell
data Point = Point Int Int deriving(Show, Read, Eq)
```

```
ghci> Point 1 2
Point 1 2
```

```
ghci> Point 1 2 /= Point 3 4
True
```

```
ghci> read "Point 1 1" :: Point
Point 1 1
```

Type Variables in Custom Types

We can use **type variables** in custom types

```
data Point a = Point a a
```

```
ghci> :t Point (1::Int) (2::Int)
Point Int
```

```
ghci> :t Point "hello" "there"
Point [Char]
```

Type Variables in Custom Types

We can use **multiple** variables in the same type

```
data Things a b c = Things a b c   deriving(Show)
```

```
ghci> Things "string" 1 True
Things "string" 1 True
```

```
ghci> Things [] 1.5 'a'
Things [] 1.5 'a'
```

Type Variables in Custom Types

We can write **functions** using these types

```
first_thing (Things x _ _) = x
```

```
ghci> first_thing (Things 1 2 3)
1
```

```
ghci> :t first_thing
first_thing :: Things a b c -> a
```

Case expressions

`case` expressions can do pattern matching **in functions**

```
data Example = One Int | Two Float

f :: Example -> Int
f x = case x of One a -> a
                Two b -> 0
```

```
ghci> f (One 3)
3
ghci> f (Two 4.0)
0
```

Case expressions

The **syntax** for a case expression is

```
case [expression] of [pattern 1] -> [expression]
                     [pattern 2] -> [expression]
                     ...
                     [pattern k] -> [expression]
```

You can use _ (the wildcard) as a catch-all

Case expressions

You can write all the patterns on **one line**

```
case x of {One a -> a; Two b -> 0}
```

Case is an **expression**

```
ghci> (case 1 of 1 -> 1) + (case 2 of 2 -> 1)
2
```

Exercises

1. Create a parameterized custom type `BigThings` that can store four things, each of which have different types. In ghci, create a value of type `BigThings`

2. Write a function `middleTwo` that takes a `BigThings` and returns the middle two elements in a tuple

3. Use the declaration `data Example = One Int | Two Float` to write a function `isOne :: Example -> Bool` that returns `True` for a value with a `One` constructor, and `False` otherwise. Use the `case` syntax to do this