Algebraic data types and type classes

1 Adding new data types

1.1 Defining data types

- So far we've only used the builtin data types
- Of course, it often makes sense to define new data types
- Multiple reasons to do this
  - Hide complexity
  - Build new abstractions
  - Type safety
- Haskell has three ways to do this
  - **type**
  - **data**
  - **newtype** (won't cover this)

1.2 Type declarations

A new name for an existing type can be defined using a type declaration

```
type String = [Char]
```

We can use these type declarations to make the semantics of our code cleaner

```
type Pos  = (Int, Int)
origin :: Pos
origin :: (0,0)
left :: Pos -> Pos
left (i,j) = (i-1,j)
```

Just like function definitions, type declarations can be parametrised over type variables

```
type Pair a = (a,a)
mult :: Pair Int -> Int
mult (m, n) = m*n

dup :: a -> Pair a
dup x  = (x,x)
```

You can't have class constraints in the definition and can't use recursive types

```
-- Not allowed
type Tree = (Int, [Tree])
```

1.3 Data Declarations: New types

We can introduce a completely new type by specifying allowed values using a data declaration

```
data Bool = False | True
```

Both the type name, and the constructor names, must begin with an upper-case letter

1.4 Using new types

```
data IsTrue = Yes | No | Perhaps
negate :: IsTrue -> Is True
negate Yes     = No
negate No      = Yes
negate Perhaps = Perhaps
```

1.5 Data declarations with fixed type parameters

The constructors in a data declaration can take arbitrarily many parameters

```
data Shape = Circle Float | Rectangle Float Float
```

"A shape is either a Circle, or a Rectangle. The circle is defined by one number, the rectangle by two"

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle x y) = x * y
```

1.6 Recursive types

Data declarations can refer to themselves

```
data Nat = Zero | Succ Nat
```

"Nat is a new type with constructors `Zero :: Nat` and `Succ :: Nat -> Nat`"

This type has an infinite set of values

We could use this to implement a representation of the natural numbers, and arithmetic

This allows for very succinct definitions of data structures

```
data List a = Empty | Const a (List a)
intList = Cons 1 (Cons 2 ( Cons 3 Empty))
== [1,2,3]
```

1.7 Some type theory and contrasts

Haskell's data declarations make Algebraic data types
This is a type where we specify the "shape" of each element

The two algebraic operations are "sum" and "product"

Definition: Sum type

An alternation

```
data Foo = A|B
```

The value of type Foo can either be A or B, example of this is `Bool`

Definition: Product type

A combination

```
data Pair = P Int Double
```

Provide a pair of numbers, an `Int` and `Double` together

1.8 Haskell Types: Pros and Cons

Classes	Algebraic Data Types
✓ Easy to add new "kinds of things": just make a subclass	x Hard to add new "kinds of things": need to add new constructor and update all functions that use the data type
x Hard to add new "operation on existing things": need to change superclass to add new method and potentially update all subclasses	✓ Easy to add new "operation on existing things": just write a new function

2 Higher order functions and type classes again

2.1 Separating code and data

- When designing software, a good aim is to hide the implementation of data structures
- In OO based languages we do this with classes and inheritence
- Or with interfaces, which define a contract that a class must implement
- Idea is that calling code doesn't know internals and only relies on interface
- As a result, we can change the implementation, and the client code still works

2.2 Generic higher order functions

- In Haskell we can realise this idea with generic higher order functions, and type classes
- Last time, we saw some examples of higher order functions for list
- For example, imagine we want to add two lists pairwise

```haskell
-- By hand
addLists _[] = []
addLists [] _ = []
addLists (x:xs) (y:ys) = (x+y) : addLists xs ys
-- Better
addLists xs ys = map (uncurry (+)) $ zip xs ys
-- Best
addLists = zipWith (+)
```

- If we write our own data types, are we reduced to doing everything "by hand" again

2.3 Using type classes

- Recall, Haskell has a concept of type classes
- These describe interfaces that can be used to constrain the polymorphism of functions to those types satisfying the interface
- (+) acts on any type, as long as that type implements the Num interface

```haskell
(+) :: Num a => a -> a -> a
```

- (<) acts on any type, as long as that type implements the Ord(erable) interface

```haskell
(<) :: Ord a => a -> a -> Bool
```

- Haskell comes with many such type classes encapsulating common patterns
- When we implement out own data types, we can "just" implement appropriate instances of these classes