

User-Defined Types

- So far we have only used the predefined types of Haskell, sometimes renaming them with synonyms:
`Char`, `Int`, `[Int]`, etc.
`type Date = (Day,Month,Year)`, `type Poly = [Int]`, etc.
- For representing “real-world” data types, we can do a lot with just those types
- However, some “real-world” data types do not naturally correspond to standard types and would require some kind of coding
- For example, to represent the months of the year or the days of the week we could use the integers 1–12 and 1–7 respectively
- But in that case it is not clear if the literal 3 represents March, Tuesday, Wednesday, the value 3, etc.

User-Defined Types

- Other examples where data type “coding” is inconvenient, unclear or unnatural:
 - a type which is *either a number or a string* (for example, in certain areas houses may have names rather than numbers)
 - a tree data structure
- Any type can be represented using the predefined types, but the representation may not be very natural
- Haskell *user-defined types* or *algebraic types* let us define new types to more naturally model types like those above

User-Defined Types

- User-defined types can be defined through the use of datatype declarations, which have the following format:

```
data Typename
  = Con1 t11...t1k1
  | ...
  | Conn tn1...tnkn
```

- These can be used to create the following types:
 - enumerated types
 - composite types
 - recursive types
 - parametric types

Enumerated Types

- A data type which consists of a finite number of constants is called an *enumerated type*
- For example, the days of the week and the months of the year could be defined as follows:

```
data Days = Mon|Tue|Wed|Thu|Fri|Sat|Sun
data Days =
Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec
```

- Other examples:

```
data Temp = Hot|Cold
data Season = Spring|Summer|Autumn|Winter
```

Enumerated Types

```
data Temp = Hot|Cold
data Season = Spring|Summer|Autumn|Winter
```

- These are new types
- Type `Temp` consists of only two values: `Hot` and `Cold`
- `Hot` and `Cold` are called *constructors* of type `Temp`
- Constructors must begin with a capital letter (to distinguish them from variables)
- Remember that type names must also begin with a capital letter

Enumerated Types

- Constructors may appear in patterns
- This is the standard way of defining functions over algebraic types

- For example:

```
weather Season -> Temp
weather Summer = Hot
weather _ = Cold
```

- Algebraic types do not automatically have operators such as equality, ordering, show, etc.
- For example, consider the following definition:

```
weather s
  | s == Summer = Hot
  | otherwise = Cold
```

- This gives the following error message:
Season is not an instance of class ‘Eq’

Enumerated Types

- The function `==` is not defined over the algebraic type `Season`
- If you want some overloaded operator such as `==` to apply to a user-defined algebraic type, you need to declare the type as an instance of the class over which the operator is defined, *ie.* equality class `Eq`
- Haskell has a standard way of deriving instances for algebraic types
- For example:

```
data Season = Spring|Summer|Autumn|Winter
           deriving Eq
```

Enumerated Types

- If we want to show the new values as strings, we also need to derive an instance of class `show` otherwise:

```
> weather Autumn
```

```
Error: Cannot find 'show' function for:
```

```
*** expression: weather Autumn
```

```
*** of type: Temp
```

- We need:

```
data Season = Spring|Summer|Autumn|Winter
             deriving (Eq, Show)
```

```
data Temp = Hot|Cold
           deriving (Eq, Show)
```

- Often we also want to derive an instance of classes `Ord` and `Enum` so that we can use the relational operators and progressions such as `[Mon..Fri]`

Function Declaration and Classes

- Consider the following function which decides whether three integers are equal:

```
allEqual :: Int -> Int -> Int -> Bool
allEqual m n p = (m==n) && (n==p)
```

- The definition has nothing which is specific to integers
- The only constraint is that m , n and p are compared for equality, *ie.* their type is an instance of class `Eq`
- So their type can be a for *any* a in the type class `Eq`
- This gives `allEqual` a most general type:

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual m n p = (m==n) && (n==p)
```

Function Declaration and Classes

```
allEqual :: Eq a => a -> a -> a -> Bool
allEqual m n p = (m==n) && (n==p)
```

- The part before `=>` (*ie.* `Eq a`) is called the *context*
- We can read the type declaration as saying that:
if the type `a` is in the class `Eq` – that is if `==` is defined over the type `a` – then `allEqual` has type `a -> a -> a -> Bool`
- This means that `allEqual` can be used with the following types:

```
allEqual :: Char -> Char -> Char -> Bool
allEqual :: Float -> Float -> Float -> Bool
allEqual :: Season -> Season -> Season -> Bool
```
- `Char`, `Float`, `Season` belong to `Eq` among many other types

Composite Types

- A composite type is a type whose values contain values of other types
- The alternatives in data definition can include *other types*, rather than being simple constants like `Hot` or `Cold`:

```
data Shape =  
    Circle Float | Rectangle Float Float
```
- `Shape` is a new type and `Circle` and `Rectangle` are *constructor functions*
 - `Circle` has the type `Float -> Shape`
 - `Rectangle` has the type `Float -> Float -> Shape`
- In other words, `Circle` *constructs* `Shape` from any `Float` value

Composite Types

- Constructor functions are the only functions that can appear in patterns
- For example:

```
isRound :: Shape -> Bool
isRound (Circle r) = True
isRound (Rectangle l b) = False
```

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