Chapter 10

New Kinds of Data

So far, we have considered types for numbers, and others like **Bool** and **Char**. We have seen the compound types of lists and tuples. We have built functions from and to these types. It would be possible to encode anything we wanted as lists and tuples of these types, but it would lead to complex and error-strewn programs. It is time to make our own types. New types are introduced using **`data`**. Here is a type for colours:

```
GHCi:
Prelude> data Colour = Red | Green | Blue | Yellow
Prelude> :type Red
Red :: Colour
```

The name of our new type is Colour (it must have a capital letter). It has four *constructors*, written with initial capital letters: `Red`, `Green`, `Blue`, and `Yellow`. These are the possible forms a value of type colour may take. Now we can build values of type Colour:

```
col :: Colour
cols :: [Colour]
colpair :: (Char, Colour)

col = Blue

cols = [Red, Red, Green, Yellow]

colpair = ('R', Red)
```

We notice an immediate problem, though. Haskell does not seem to know how to print out the values of our new type:

```
GHCi:
Prelude> data Colour = Red | Green | Blue | Yellow
Prelude> col = Blue
```

```
Prelude> col

<interactive>:4:1: error:
    • No instance for (Show Colour) arising from a use of 'print'
    • In a stmt of an interactive GHCi command: print it
```

If we are happy for Haskell to try to work out automatically how to print values of type Colour we can use write **deriving** Show after the type:

```
Prelude> data Colour = Red | Green | Blue | Yellow deriving Show
Prelude> col = Blue
Prelude> col
Blue
```

The word **deriving** tells Haskell to try to work out automatically (to derive) how to make Colour an instance of the type class **Show**, which is the class of all types which can be automatically printed out. Most of the types we have seen so far (booleans, numbers, lists, tuples, but not functions) are instances of the typeclass **Show**, which is why Haskell can print them. The reason we do not see, for example (**Num** a, **Show** a) ⇒ a for a number is that, since **Num** is a *subclass* of **Show**, we need only write **Num**, and **Show** is implied. Now that we have used **deriving**, Colour is also an instance of **Show**. In fact, the built-in function to show any showable thing, which Haskell uses, is called `show` and we can use it ourselves:

```
GHCi:
Prelude> show [1, 2, 3]
"[1,2,3]"
```

What is the type of the built-in `show` function? It is **Show** a ⇒ a → **String** because, given anything which is showable, it produces a **String** representing it.

Type definitions may contain a *type variable* like a to allow the type of part of the new type to vary – i.e. for the type to be polymorphic. For example, we can define the **Maybe** type ourselves.

```
data Maybe a = Nothing | Just a deriving Show
```

In addition to being polymorphic, new types may also be recursively defined. We can use this functionality to define our own lists, just like the built-in lists in Haskell but without the special notation:

```
data Sequence a = Nil | Cons a (Sequence a) deriving Show
```

We have called our type Sequence to avoid confusion. It has two constructors: `Nil` which is equivalent to `[]`, and `Cons` which is equivalent to the `:` operator. `Cons` carries two pieces of data with it – one of type a (the head) and one of type Sequence a (the tail). This is the recursive part of our definition. Now we can make our own lists equivalent to Haskell's built-in ones:

Built-in	Ours	Our Type
`[]`	`Nil`	Sequence a
`['a', 'x', 'e']`	`Cons 'a' (Cons 'x' (Cons 'e' Nil))`	Sequence **Char**
`[1]`	`Cons 1 Nil`	**Num** a ⇒ Sequence a
`[Red, RGB 2 2 2]`	`Cons (Red, Cons (RGB (2, 2, 2), Nil))`	**Num** a ⇒ Sequence (Colour a)

Now you can see why getting at the last element of a list in Haskell is harder than getting at the first element – it is deeper in the structure. Let us compare some functions on Haskell lists with the same ones on our new Sequence type. First, the ones for built-in lists.

```
length' :: Num b ⇒ [a] → b
append :: [a] → [a] → [a]

length' [] = 0
length' (_:xs) = 1 + length' xs

append [] ys = ys
append (x:xs) ys = x : append xs ys
```

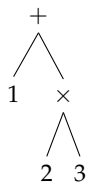
And now the same functions with our new Sequence type:

```
seqLength :: Num b ⇒ Sequence a → b
seqAppend :: Sequence a → Sequence a → Sequence a

seqLength Nil = 0
seqLength (Cons _ xs) = 1 + seqLength xs

seqAppend Nil ys = ys
seqAppend (Cons x xs) ys = Cons x (seqAppend xs ys)
```

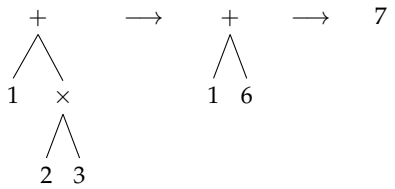
Notice how all the conveniences of pattern matching such as the use of the underscore work for our own types too.

A Type for Mathematical Expressions

Our Sequence was an example of a recursively-defined type, which can be processed naturally by recursive functions. Mathematical expressions can be modeled in the same way. For example, the expression $1 + 2 \times 3$ could be drawn like this:



Notice that, in this representation, we never need parentheses – the diagram is unambiguous. We can evaluate the expression by reducing each part in turn:



Here is a suitable type for such expressions:

```
data Expr a = Num a
            | Add (Expr a) (Expr a)
            | Subtract (Expr a) (Expr a)
            | Multiply (Expr a) (Expr a)
            | Divide (Expr a) (Expr a) deriving Show
```

For example, the expression $1 + 2 * 3$ is represented in this data type as:

```
Add (Num 1) (Multiply (Num 2) (Num 3))
```

We can now write a function to evaluate expressions:

```
evaluate :: Integral a ⇒ Expr a → a

evaluate (Num x) = x
evaluate (Add e e') = evaluate e + evaluate e'
evaluate (Subtract e e') = evaluate e - evaluate e'
evaluate (Multiply e e') = evaluate e * evaluate e'
evaluate (Divide e e') = evaluate e `div` evaluate e'
```

Building our own types leads to clearer programs with more predictable behaviour, and helps us to think about a problem – often the functions are easy to write once we have decided on appropriate types.

Questions

1. Design a new type `Rect a` for representing rectangles. Treat squares as a special case.
2. Now write a function of type **Num** `a` ⇒ `Rect a` → `a` to calculate the area of a given `Rect`.
3. Write a function which rotates a `Rect` such that it is at least as tall as it is wide.
4. Use this function to write one which, given a `[Rect]`, returns another such list which has the smallest total width and whose members are sorted narrowest first.
5. Write versions of the `seqTake`, `seqDrop`, and `seqMap` functions for the `Sequence` type.
6. Extend the `Expr` type and the `evaluate` function to allow raising a number to a power.
7. Use the **Maybe** type to deal with the problem that a division by zero may occur in the `evaluate` function.