

Data Types

So far, we have used simple data types like `Int` and `Bool`, or data types that I have supplied, like `Language` or `Image`. It is time that we look at how we can define our own more complex data types.

Suppose we wish to create a data type to represent geometric shapes so that users of these types can create instances of these shapes and compute their areas. Our first task is to create a data type to represent different types of shapes, such as a rectangle or an ellipse.

```
data Shape = Rectangle Side Side |
            Ellipse Radius Radius
            deriving Show

type Radius = Float
type Side = Float
```

The `data` keyword says that we are defining a new data type that can have several options. We are naming the data type `Shape`. We now list the options separated by the `|` character. Each option is given a name of our choosing, and in this case we decided to name the options `Rectangle` and `Ellipse`. For each of these options we specify the types of the parameters. For `Rectangle` the two parameters are floats representing the length of the two sides. We could have used the code `Rectangle Float Float` but to make it more readable we instead define a synonym `Side` which we specify as a `Float` using the `type` keyword. The `type` keyword simply allows us to give a data type a second name. We then define the `Ellipse` option in a similar way. We are now able to define shapes that are either rectangles or ellipses

We can define a rectangle `r1` with one side of length 2.0 and another of length 3.5:

```
r1 = Rectangle 2.0 3.5
```

We could define functions for creating squares and circles:

```
square s = Rectangle s s

circle r = Ellipse r r
```

We can define an `area` function which will compute the area for any of our shapes. To do this we give the definition for each of the possible options:

```
area :: Shape -> Float
area (Rectangle s1 s2) = s1 * s2
area (Ellipse r1 r2) = pi * r1 * r2
```

We can now apply the **area** function to any of our previously defined shapes:

```
area r1
```

Haskell uses a built-in pattern matching mechanism on the argument of the function to check whether it is either a rectangle or an ellipse in order to use the correct option in our function definition.