Custom Types and the Option Type

In this section we discuss one of the most important features of Haskell and many functional programming languages, namely the ability to create new types. In this section we explore simple cases of new types. Later on we will look at the quite powerful recursive types, that allow us to define recursive structures.

Reading

- Sections 8.1-8.3

Type Aliases and Custom Types

One of the most important features of Haskell is the ability to create new types from existing types. They are an important way of organizing our programs, by defining the kinds of values that our programs would operate on.

There are fundamentally two ways of producing new types: **type aliases** and **custom data types**.

Type Aliases

Type aliases are simply new names we give to existing types. For example, we can define 2-dimensional and 3-dimensional point types as aliases of tuple types:

```
type Point2D = (Double, Double)
type Point3D = (Double, Double, Double)
```

We can then use Point2D and Point3D in any place where we would use the corresponding tuple types. The two are indistinguishable, hence the use of the word *alias*.

Type aliases may also contain parametric types. For instance, we can create an "associative list", which is a list containing key-value pairs, like so:

```
type Assoc k v = [(k, v)]
```

Here k and v are parametric types.

Custom Data Types

Custom data types are what is often refered to as a "union type". In a custom data type we state that a value of this type can be any of a number of alternatives, all differentiated by a keyword called a "constructor". The simplest example of a union type is in fact the Bool type, which we can define as follows:

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

So a custom data type starts with the keyword data instead of type, and it is followed by the type name, in this case Bool. After the equals sign we offer the various alternative forms that a value of this type can take, separated by vertical lines. In this example, there are exactly two possibilities, True and False.

As another example, in a card game we could specify a type representing the card's suit:

```
data Suit = Clubs | Diamonds | Hearts | Spades
```

We could then define a card as:

```
type Card = (Int, Suit)
```

And create cards like (5, Diamonds).

Deriving type classes for data types   Haskell can automatically generate standard implementations for many data types, if we ask it to. We do this by adding the keyword deriving at the end of the definition, like so:

```
data Bool = False | True    deriving (Eq, Ord, Show)
```

It does so in a straightforward way: Two values are equal only if they are exactly the same, the values are ordered from left to right in their definition (so True > False) and they turn to their corresponding strings under the show function.

Data types as union types   What we have seen above is the use of data types to define what is typically known as **enumerations**. It is effectively a sequence of possible values, that we could have represented by distinct numbers but which are more meaningful with these words. In C one would typically use constants for a similar purpose, but without the benefits of automatic type-checking: if True was simply another way of saying 1, then if a program ended up saying if 1 then … else … then we wouldn't know if that is because we really wanted 1 there to represent truthiness or if we made a mistake.

> Enumerations force type-checking and can prevent many hard-to-detect errors.

We will now discuss another important use of data types, with similar motivations. This is the full use of data types to represent **union types**. As an example, suppose that we wanted to write a program that can work with temperatures. We want the system to be able to work with both Fahrenheit and Celsius temperatures at the same time. One way to do this, with some problems that we will discuss in a moment, is to use a tuple type like so:

```
data TempScale = F | C          deriving (Eq, Show)
type Temp = (TempScale, Double)
```

So then we could have temperatures like (F, 56) for "56 Fahrenheit" and so on. We could then write a function:

```
toFahrenheit :: Temp -> Double
toFahrenheit (F, fTemp) = fTemp
toFahrenheit (C, cTemp) = 9 / 5 * cTemp + 32
```

This can work, but it has a number of subtle problems. The main problem is that there is no way to guarantee that a function would account for both temperatures. Someone could write a toFahrenheit function like so:

```
toFahrenheit (_, temp) = temp
```

which is of course logically wrong, but the type system does not prevent one from doing so.

Data types offer us a different approach that forces us to handle the different temperatures. Instead of representing a temperature as a tuple, we represent it as a double with a "tag" in front of it to distinguish between F or C. It would look like this:

```
data Temp = F Double | C Double    deriving (Show)
```

Then we can write a temperature value as F 23 or as C 12.3 and so on. The scale tag is now part of the double value, and you cannot look at that value without discussing the tag. To take the value out we need to do a pattern-match, and we are forced to have both F and C branches:

```
toFahrenheit :: Temp -> Double
toFahrenheit (F fTemp) = fTemp
toFahrenheit (C cTemp) = 9 / 5 * cTemp + 32
```

We have just seen a new form of pattern-matching. Using a data-type's constructor (here F and C) along with a pattern for their contents.

As another example, suppose we wanted to do some arithmetic with fractions. We want to allow two kinds of numbers: integers and fractions. We want the two kinds of numbers to coexist in one type. We can represent fractions as integer pairs: (Int, Int). The question is how to handle the intgers. One option would be to force each integer n into the fraction n / 1. But a more expressive option is to use a union type:

```
data Number = NumInt Int | NumFrac (Int, Int)
```

Then we could write a multiply function, that looks like this (without worrying about simplifying the fraction):

```
mult :: Number -> Number -> Number
NumInt n 'mult' NumInt m                = NumInt (n * m)
NumInt n 'mult' NumFrac (top, bottom)   = NumFrac (n * top, bottom)
NumFrac (top, bottom) 'mult' NumInt n   = NumFrac (n * top, bottom)
NumFrac (t1, b1) 'mult' NumFrac (t2, b2) = NumFrac (t1 * t2, b1 * b2)
```

The Maybe option type

One particular built-in datatype deserves special mention. It is known as the Maybe type in Haskell, and also as the Option type in other languages. Its premise is simple: It allows you to carry one value around, but also allows the possibility of no value

at all. For example, imagine we write a `lookup` method that looks for an key in an associative list:

```
lookup :: Eq a => a -> [(a, b)] -> b
```

This function is meant to search for the `a` part of the pair in the list, and if it finds it then it returns the corresponding `b` part. But what should happen if a suitable `a` part is not found? What value would the function return?

In other languages we have something like a `null` value. This is highly problematic, for a reason similar to the one described in temperatures.

> Using the `null` value to indicate failure, we have no way of expressing in our type system whether a function may return `null` for its result, and whether a function should be handling `null` as a possible input to one of its arguments. These functions all look alike as far as their signature is concerned.

So for example `lookup` has no way of telling its users "hey my answer may be null so you better handle that possibility". Similarly the typechecker has no way of knowing if a function that uses the result of lookup bothers to check for the null case.

Option types solve these problems for us. The `Maybe` type is defined as follows:

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

So a value of type `Maybe Int` is either `Nothing` or something like `Just 5`. Then the (proper) type for the `lookup` function above is (for completeness we include its implementation):

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup key []                      = Nothing
lookup key ((target, value) : rest)
    | key == target                = Just value
    | otherwise                    = lookup key rest
```

Now anyone who wants to use the result of the lookup must do a pattern match on the two different forms, and as a result somehow handle the `Nothing` case.

```
reportLookupResult :: (Eq a, Show b) => a -> [(a, b)] -> String
reportLookupResult key lst = handleResult (lookup key lst)
    where handleResult Nothing = "No matches found!"
          handleResult (Just someb) = "Found one: " ++ show someb
```

A standard example of this is a "safe division" function, which does not allow you to divide by 0. It would look like this:

```
safeDivide :: Num t => t -> t -> Maybe t
safeDivide _ 0  = Nothing
safeDivide n m  = Just (n / m)
```

Standard functions for Maybe   There are a number of standard functions for the Maybe type. We declare them here, and ask you to implement them for practice:

— Empty or one–element list
toList :: **Maybe** a –> [a]
— Apply the function to the a if there is one, otherwise just pass the default b
maybe :: b –> (a –> b) –> **Maybe** a –> b
— Preserve the Nothing or apply the function
fmap :: (a –> b) –> **Maybe** a –> **Maybe** b
— Preserve the Nothing or apply the function
(>>=) :: **Maybe** a –> (a –> **Maybe** b) –> **Maybe** b
— Preserve the Nothing or keep the b value
(>>) :: **Maybe** a –> **Maybe** b –> **Maybe** b
— Wrap a value in Maybe
return :: a –> **Maybe** a
— Preserve the Nothing or apply the function
(<∗>) :: **Maybe** (a –> b) –> **Maybe** a –> **Maybe** b
— Apply the function if possible
foldr :: (a –> b –> b) –> b –> **Maybe** a –> b

Practice: Implement the above functions.