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)
```

Practice:

1. The uncons function is meant to take a list and "uncons" it, which means to return the pair of the first element (head) and the rest of the list (tail). This of course is only valid if the list is nonempty. Using Maybe makes the function work always:

   ```
   uncons :: [a] -> Maybe (a, [a])
   ```

   Implement it.

2. Write the type and implementation for a safeMax function, which returns the maximum of a list (or Nothing if there is no maximum, i.e. if the list is empty). Your list element values only need to be comparable, so use the appropriate type class constraint.

3. Write a function unmaybe :: [Maybe a] -> [a] that takes a list of Maybe a's and returns a list of the actual values that are there, if any, in the same order.

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.