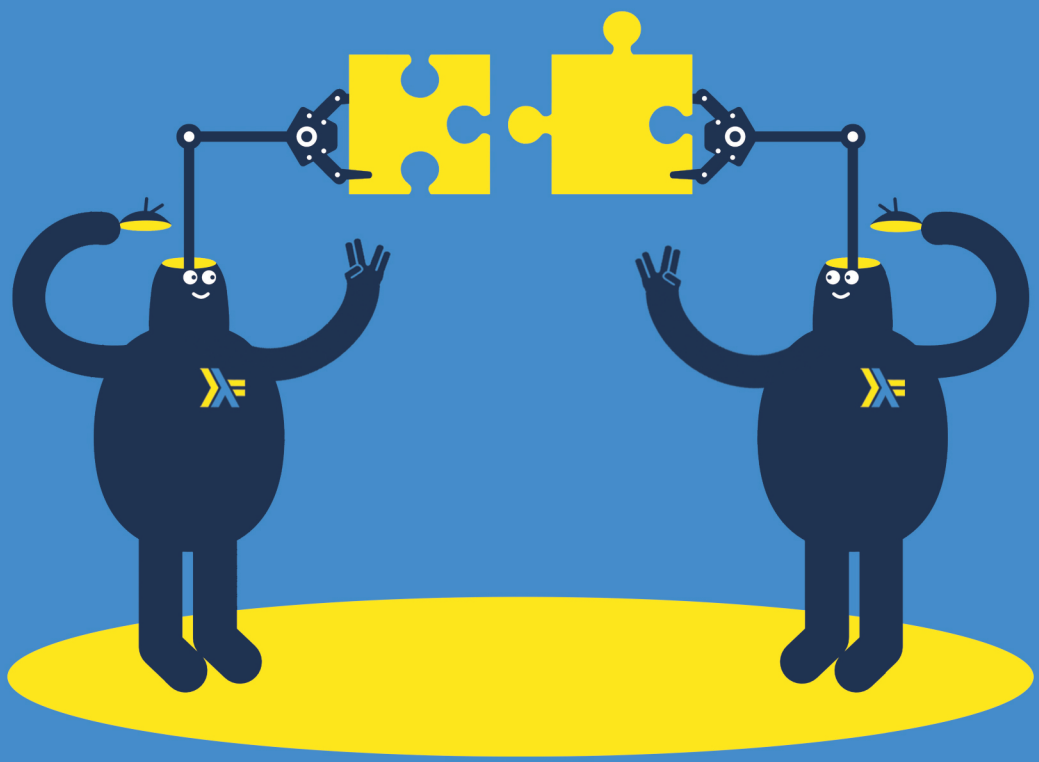
GET PROGRAMMING WITH HASKELL



Will Kurt

MANNING

Get Programming with Haskell

by Will Kurt

Lesson 27

Contents

Unit 6

ORGANIZING CODE AND BUILDING PROJECTS

Unit 7

PRACTICAL HASKELL

UNIT

5

Working with type in a context

In this unit, you'll take a look at three of Haskell's most powerful and often most confusing type classes: `Functor`, `Applicative`, and `Monad`. These type classes have funny names but a relatively straightforward purpose. Each one builds on the other to allow you to work in contexts such as `IO`. In unit 4, you made heavy use of the `Monad` type class to work in `IO`. In this unit, you'll get a much deeper understanding of how that works. To get a better feel for what these abstract type classes are doing, you'll explore types as though they were shapes.

One way to understand functions is as a means of transforming one type into another. Let's visualize two types as two shapes, a circle and a square, as shown in figure 1.

Figure 1 A circle and square visually representing two types

These shapes can represent any two types, `Int` and `Double`, `String` and `Text`, `Name` and `FirstName`, and so forth. When you want to transform a circle into a square, you use a function. You can visualize a function as a connector between two shapes, as shown in figure 2.

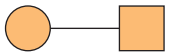


Figure 2 A function can transform a circle to a square.

This connector can represent any function from one type to another. This shape could represent (`Int -> Double`), (`String -> Text`), (`Name -> FirstName`), and so forth. When you want to apply a transformation, you can visualize placing your connector between the initial shape (in this case, a circle) and the desired shape (a square); see figure 3.

Figure 3 Visualizing a function as a way of connecting one shape to another

As long as each shape matches correctly, you can achieve your desired transformation.

In this unit, you'll look at working with types in context. The two best examples of types in context that you've seen are `Maybe` types and `IO` types. `Maybe` types represent a context in which a value might be missing, and `IO` types represent a context in which the value has interacted with I/O. Keeping with our visual language, you can imagine types in a context as shown in figure 4.

Figure 4 The shape around the shape represents a context, such as `Maybe` or `IO`.

These shapes can represent types such as `IO Int` and `IO Double`, `Maybe String` and `Maybe Text`, or `Maybe Name` and `Maybe FirstName`. Because these types are in a context, you can't simply use your old connector to make the transformation. So far in this book, you've relied on using functions that have both their input and output in a context as well. To perform the transformation of your types in a context, you need a connector that looks like figure 5.

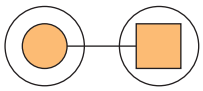


Figure 5 A function that connects two types in a context

This connector represents functions with type signatures such as (`Maybe Int -> Maybe Double`), (`IO String -> IO Text`), and (`IO Name -> IO FirstName`). With this connector, you can easily transform types in a context, as shown in figure 6.

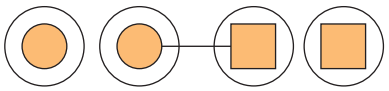


Figure 6 As long as your connector matches, you can make the transformation you want.

This may seem like a perfect solution, but there's a problem. Let's look at a function `halve`, which is of the type `Int -> Double`, and as expected halves the `Int` argument.

Listing 1 A `halve` function from `Int -> Double`

```
halve :: Int -> Double
halve n = fromIntegral n / 2.0
```

This is a straightforward function, but suppose you want to halve a `Maybe Int`. Given the tools you have, you have to write a wrapper for this that works with `Maybe` types.

Listing 2 `halveMaybe` wraps `halve` function to work with `Maybe` types

```
halveMaybe :: Maybe Int -> Maybe Double
halveMaybe (Just n) = Just (halve n)
halveMaybe Nothing = Nothing
```

For this one example, it's not a big deal to write a simple wrapper. But consider the wide range of existing functions from `a -> b`. To use any of these with `Maybe` types would require nearly identical wrappers. Even worse is that you have no way of writing these wrappers for `IO` types!

This is where `Functor`, `Applicative`, and `Monad` come in. You can think of these type classes as adapters that allow you to work with different connectors so long as the underlying types (circle and square) are the same. In the `halve` example, you worried about transforming your basic `Int`-to-`Double` adapter to work with types in context. This is the job of the `Functor` type class, illustrated in figure 7.

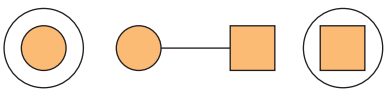


Figure 7 The `Functor` type class solves this mismatch between types in a context and a connector.

But you can have three other types of mismatches. The `Applicative` type class solves two of these. The first occurs when the first part of your connector is in a context, but not its result, as shown in figure 8.

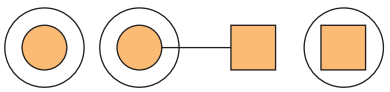


Figure 8 This is one of the mismatches that Applicative solves.

The other problem occurs when an entire function is in a context. For example, a function of the type `Maybe (Int -> Double)` means you have a function that might itself be missing. This may sound strange, but it can easily happen when using partial application with `Maybe` or `IO` types. Figure 9 illustrates this interesting case.

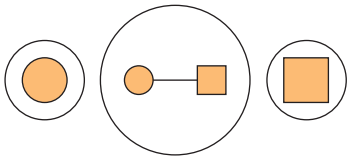


Figure 9 Sometimes the connector itself is trapped in a context; `Applicative` solves this problem as well.

There's only one possible mismatch between a function and types in a context left. This occurs when the argument to a function isn't in a context, but the result is. This is more common than you may think. Both `Map.lookup` and `putStrLn` have type signatures like this. This problem is solved by the `Monad` type class, illustrated in figure 10.

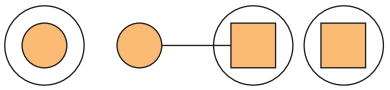


Figure 10 The `Monad` type class provides an adapter for this final possible mismatch.

When you combine all three of these type classes, there's no function that you can't use in a context such as `Maybe` or `IO`, so long as the underlying types match. This is a big deal because it means that you can perform any computation you'd like in a context and have the tools to reuse large amounts of existing code between different contexts.

27 LESSON

THE FUNCTOR TYPE CLASS

After reading lesson 27, you'll be able to

- Use the `Functor` type class
- Solve problems with `fmap` and `<$>`
- Understand kinds for `Functors`

So far in this book, you've seen quite a few parameterized types (types that take another type as an argument). You've looked at types that represent containers, such as `List` and `Map`. You've also seen parameterized types that represent a context, such as `Maybe` for missing values and `IO` for values that come from the complex world of I/O. In this lesson, you'll explore the powerful `Functor` type class. The `Functor` type class provides a generic interface for applying functions to values in a container or context. To get a sense of this, suppose you have the following types:

- `[Int]`
- `Map String Int`
- `Maybe Int`
- `IO Int`

These are four different types, but they're all parameterized by the same type: `Int` (`Map` is a special case, but the values are type `Int`). Now suppose you have a function with the following type signature:

```
Int -> String
```

This is a function that takes an `Int` and returns a `String`. In most programming languages, you'd need to write a custom version for your `Int -> String` function for each of these parameterized types. Because of the `Functor` type class, you have a uniform way to apply your single function to all these cases.

Consider this You have a potentially missing `Int` (a `Maybe Int`). You want to square this value, turn it into a string, and then add an `!` to the end. The function that you want to pass this value to, `printInt`, assumes that there might be missing values already:

```
printInt :: Maybe String -> IO ()
printInt Nothing = putStrLn "value missing"
printInt (Just val) = putStrLn val
```

How can you transform your `Maybe Int` into a `Maybe String` to be used by `printInt`?

27.1 An example: computing in a Maybe

`Maybe` has already proven a useful solution to your problem of potentially missing values. But when you were introduced to `Maybe` in lesson 19, you still had to deal with the problem of handling the possibility of a missing value as soon as you encountered it in your program. It turns out you can do computation on a potentially missing value without having to worry about whether it's actually missing.

Suppose you get a number from a database. There are plenty of reasons why a request to a database would result in a null value. Here are two sample values of type `Maybe Int`: `failedRequest` and `successfulRequest`.

Listing 27.1 Possibly null values: `successfulRequest` and `failedRequest`

```
successfulRequest :: Maybe Int
successfulRequest = Just 6

failedRequest :: Maybe Int
failedRequest = Nothing
```

Next imagine you need to increment the number you received from the database and then write it back to the database. From a design standpoint, assume that the logic that talks to the database handles the case of null values by not writing the value. Ideally,

you'd like to keep your value in a `Maybe`. Given what you know so far, you could write a special `incMaybe` function to handle this.

Listing 27.2 Defining `incMaybe` to increment `Maybe Int` values

```
incMaybe :: Maybe Int -> Maybe Int
incMaybe (Just n) = Just (n + 1)
incMaybe Nothing = Nothing
```

In GHCi, this works just fine:

```
GHCi> incMaybe successfulRequest
Just 7
GHCi> incMaybe failedRequest
Nothing
```

The problem is that this solution scales horribly. The increment function is just `(+ 1)`, but in our example, you need to rewrite it for `Maybe`. This solution means that you'd have to rewrite a special version of every existing function you want to use in a `Maybe`! This greatly limits the usefulness of tools such as `Maybe`. It turns out Haskell has a type class that solves this problem, called `Functor`.

Quick check 27.1 Write the function `reverseMaybe :: Maybe String -> Maybe String` that reverses a `Maybe String` and returns it as a `Maybe String`.

27.2 Using functions in context with the Functor type class

Haskell has a wonderful solution to this problem. `Maybe` is a member of the `Functor` type class. The `Functor` type class requires only one definition: `fmap`, as shown in figure 27.1.

QC 27.1 answer

```
reverseMaybe :: Maybe String -> Maybe String
reverseMaybe Nothing = Nothing
reverseMaybe (Just string) = Just (reverse string)
```

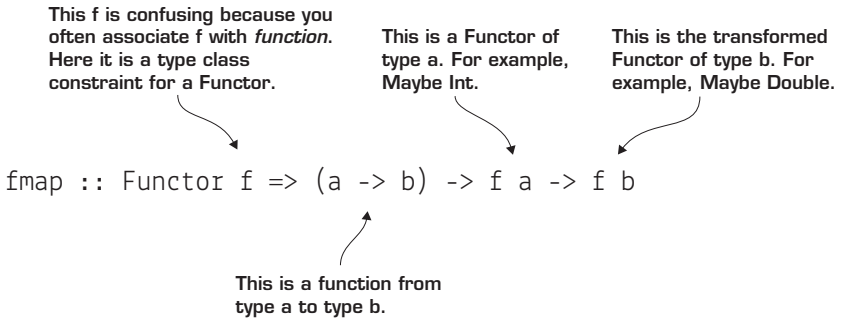


Figure 27.1 The type signature for the `fmap` function

Going back to your visual language from the introduction, `fmap` provides an adapter, as shown in figure 27.2. Notice that we're using `<$>`, which is a synonym for `fmap` (except it's a binary operator rather than a function).

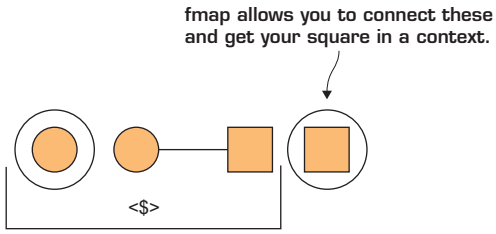


Figure 27.2 Visualizing how `fmap`, also `<$>`, works as an adapter, allowing you to work with types in a context.

You can define `fmap` as a generalization of your custom `incMaybe` function.

Listing 27.3 Making `Maybe` an instance of `Functor`

```
instance Functor Maybe where
  fmap func (Just n) = Just (func n)
  fmap func Nothing = Nothing
```

With `fmap`, you no longer need a special function for keeping your value in a `Maybe`:

```
GHCi> fmap (+ 1) successfulRequest
Just 7
GHCi> fmap (+ 1) failedRequest
Nothing
```

Though `fmap` is the official function name, in practice the binary operator `<$>` is used much more frequently:

```
GHCi> (+ 1) <$> successfulRequest
Just 7
GHCi> (+ 1) <$> failedRequest
Nothing
```

In this example, `(+ 1)` adds 1 into the `Maybe Int` and returns a `Maybe Int` as well. But it's important to realize that the type signature of the function in `fmap` is `(a -> b)`, meaning that the `Maybe` returned doesn't need to be parameterized by the same type. Here are two examples of going from a `Maybe Int` to a `Maybe String`.

Listing 27.4 Examples of using `fmaps` from one type to another

```
successStr :: Maybe String
successStr = show <$> successfulRequest

failStr :: Maybe String
failStr = show <$> failedRequest
```

This ability to transform the types of values inside a `Maybe` is the true power of the `Functor` type class.

Quick check 27.2 Use `fmap` or `<$>` to reverse a `Maybe String`.

QC 27.2 answer

```
GHCi> reverse <$> Just "cat"
Just "tac"
```

Strange type class names?

`Semigroup`, `Monoid`, and now `Functor`! What's up with these weird type class names? All these names come from fields of mathematics called abstract algebra and category theory. You absolutely don't need to know any advanced mathematics to use them. All of these type classes represent the design patterns of functional programming. If you've used Java, C#, or any enterprise programming language, you're likely familiar with object-oriented design patterns such as the Singleton, Observer, and Factory patterns. These names are more reasonable-sounding only because they've become part of the everyday vocabulary of OOP. Both OOP design patterns and category theoretic type classes abstract out common programming patterns. The only difference is that Haskell's are based on mathematical foundations, rather than ad hoc patterns discovered in code. Just as Haskell's functions derive power from their mathematical basis, so do Haskell's design patterns.

27.3 Functors are everywhere!

To understand instances of `Functor`, you'll run through some examples. Recall from lesson 18 that kinds are the types of types. Types of a kind `* -> *` are parameterized types that take just one type parameter. All `Functor`s must be of kind `* ->*`. It also turns out that many parameterized types of kind `* -> *` are instances of `Functor`.

Members of `Functor` that you've seen so far in this book include `List`, `Map`, `Maybe`, and `IO`. To demonstrate how `Functor` allows you to generalize by solving a single problem the same way in multiple parameterized types, you'll explore how working with the same data type in multiple contexts can represent different problems. Then you'll see how `Functor`'s `<$>` makes it easy to solve each of these problems in the same way. Rather than work with simple types such as `Int` or `String`, you'll work with something more complicated: a `RobotPart` data type.

27.3.1 One interface for four problems

In this example, you're going to assume that you're in the business of manufacturing robot parts. Here's the basic data type for your robot part.

Listing 27.5 `RobotPart` defined using record syntax

```
data RobotPart = RobotPart
   { name :: String
   , description :: String
```

```
   , cost :: Double
   , count :: Int
   } deriving Show
```

Here are some example robot parts you'll use in this section.

Listing 27.6 Example robot parts: `leftArm`, `rightArm`, and `robotHead`

```
leftArm :: RobotPart
leftArm  = RobotPart
   { name = "left arm"
   , description = "left arm for face punching!"
   , cost = 1000.00
   , count = 3
   }

rightArm :: RobotPart
rightArm  = RobotPart
   { name = "right arm"
   , description = "right arm for kind hand gestures"
   , cost = 1025.00
   , count = 5
   }

robotHead :: RobotPart
robotHead  = RobotPart
   { name = "robot head"
   , description = "this head looks mad"
   , cost = 5092.25
   , count = 2
   }
```

One of the most common things you'll need to do is to render the information contained in a `RobotPart` as HTML. Here's code for rendering an individual `RobotPart` as an HTML snippet.

Listing 27.7 Rendering a `RobotPart` as HTML

```
type Html = String

renderHtml :: RobotPart -> Html
renderHtml part = mconcat ["<h2>",partName, "</h2>"
                          ,"<p><h3>desc</h3>",partDesc
                          ,"</p><p><h3>cost</h3>"
                          ,partCost
                          ,"</p><p><h3>count</h3>"
                          ,partCount,"</p>"]
 where partName = name part
       partDesc = description part
       partCost = show (cost part)
       partCount = show (count part)
```

In many cases, you'll want to convert a `RobotPart` into an HTML snippet. Next you'll see four scenarios of this, using different parametrized types.

You'll start by using the `Map` type to create `partsDB`, which is your internal database of `RobotPart`s.

Listing 27.8 Your `RobotPart` "database"

```
import qualified Data.Map as Map

partsDB :: Map.Map Int RobotPart
partsDB = Map.fromList keyVals
 where keys = [1,2,3]
       vals = [leftArm,rightArm,robotHead]
       keyVals = zip keys vals
```

Remember to include this in the top of your file if you're using Map.

`Map` is a useful type for this example because it naturally involves three instances of `Functor`: it's made from a `List`, returns `Maybe` values, and is itself a `Functor`.

27.3.2 Converting a Maybe RobotPart to Maybe Html

Now suppose you have a website driven by `partsDB`. It's reasonable that you'd have a request containing an ID for a part that you wish to insert into a web page. You'll assume that an `insertSnippet IO` action will take HTML and insert it into a page's template. It's also reasonable to assume that many data models might be generating

snippets. Because any one of these models may have an error, you'll assume that `insertSnippet` accepts `Maybe Html` as its input, allowing the template engine to handle missing snippets as it sees fit. Here's the type signature of your imaginary function:

```
insertSnippet :: Maybe Html -> IO ()
```

The problem you need to solve is looking up a part and passing that part as `Maybe Html` to `insertSnippet`. Here's an example of fetching a `RobotPart` from your `partsDB`.

Listing 27.9 `partVal`: a `Maybe RobotPart` value

```
partVal :: Maybe RobotPart
partVal = Map.lookup 1 partsDB
```

Because `Maybe` is a `Functor`, you can use `<$>` to transform your `RobotPart` into HTML while remaining in a `Maybe`.

Listing 27.10 Using `<$>` to transform `RobotPart` to HTML, remaining in `Maybe`

```
partHtml :: Maybe Html
partHtml = renderHtml <$> partVal
```

You can now pass `partHtml` to `insertSnippet` easily because of `Functor`.

27.3.3 Converting a list of RobotParts to a list of HTML

Next suppose you want to create an index page of all your parts. You can get a list of parts from your `partsDB` like this.

Listing 27.11 A list of `RobotParts`

```
allParts :: [RobotPart]
allParts = map snd (Map.toList partsDB)
```

`List` is also an instance of `Functor`. In fact, `fmap` for a `List` is the regular map function you've been using since unit 1. Here's how you can apply `renderHtml` to a list of values by using `<$>`.

Listing 27.12 Transforming a list of `RobotParts` to HTML with `<$>` instead of `map`

```
allPartsHtml :: [Html]
allPartsHtml = renderHtml <$> allParts
```

Because <$> is just fmap, and for lists fmap is just map, this code is identical to the following.

Listing 27.13 The traditional way of transforming a list by using map

```
allPartsHtml :: [Html]
allPartsHtml = map renderHtml allParts
```

For lists, it's more common to use map over <$>, but it's important to realize these are identical. One way to think of the Functor type class is as "things that can be mapped over."

Quick check 27.3 Rewrite the definition of all parts to use <$> instead of map.

27.3.4 Converting a Map of RobotParts to HTML

The partsDB Map has been useful, but it turns out all you need it for is converting RobotParts to HTML. If that's the case, wouldn't it make more sense to have an htmlPartsDB so you don't have to continually convert? Because Map is an instance of Functor, you can do this easily.

Listing 27.14 Turning your partsDB into a Map of HTML rather than RobotParts

```
htmlPartsDB :: Map.Map Int Html
htmlPartsDB = renderHtml <$> partsDB
```

Now you can see that you've transformed your Map of RobotParts into a Map of Html snippets!

```
GHCi> Map.lookup 1 htmlPartsDB
Just "<h2>left arm</h2><p><h3>desc</h3>left ...
```

This example highlights just how powerful the simple interface that Functor provides can be. You can now trivially perform any transformation that you can on a RobotPart to an entire Map of robot parts.

QC 27.3 answer

```
allParts :: [RobotPart]
allParts = snd <$> Map.toList partsDB
```

The careful reader may have noticed something strange about `Map` being a `Functor`. `Map`'s kind is `* -> * -> *` because `Map` takes two type arguments, one for its keys and another for its values. Earlier we said that `Functor`s must be of kind `* -> *`, so how can this be? If you look at the behavior of `<$>` on your `partsDB`, it becomes clear. `Functor` for `Map` is concerned only about the `Map`'s values and not its keys. When `Map` is made an instance of `Functor`, you're concerned only about a single type variable, the one used for its values. So for the purposes of `Map` being a member of `Functor`, you treat it as being of kind `* -> *`. When we introduced kinds in lesson 18, they may have seemed overly abstract. But they can be useful for catching issues that arise with more advanced type classes.

27.3.5 Transforming an IO RobotPart into IO Html

Finally, you might have a `RobotPart` that comes from `IO`. You'll simulate this by using `return` to create an `IO` type of a `RobotPart`.

Listing 27.15 Simulating a `RobotPart` coming from an `IO` context

```
leftArmIO :: IO RobotPart
leftArmIO = return leftArm
```

Suppose you want to turn this into HTML so that you can write the HTML snippet to a file. By now, the pattern should start to be familiar.

Listing 27.16 Transforming

```
htmlSnippet :: IO Html
htmlSnippet = renderHtml <$> leftArmIO
```

Let's take a look at all of these transformations at once:

```
partHtml :: Maybe Html
partHtml = renderHtml <$> partVal

allPartsHtml :: [Html]
allPartsHtml = renderHtml <$> allParts

htmlPartsDB :: Map.Map Int Html
htmlPartsDB = renderHtml <$> partsDB

htmlSnippet :: IO Html
htmlSnippet = renderHtml <$> leftArmIO
```

As you can see, `Functor`'s `<$>` provides a common interface to apply any function to a value in a context. For types such as `List` and `Map`, this is a convenient way to update values in these containers. For `IO`, it's essential to be able to change values in an `IO` context, because you can't take `IO` values out of their context.

Summary

In this lesson, our objective was to introduce you to the `Functor` type class. The `Functor` type class allows you to apply an ordinary function to values inside a container (for example, `List`) or a context (for example, `IO` or `Maybe`). If you have a function `Int -> Double` and a value `Maybe Int`, you can use `Functor`'s `fmap` (or the `<$>` operator) to apply the `Int -> Double` function to the `Maybe Int` value, resulting in a `Maybe Double` value. `Functor`s are incredibly useful because they allow you to reuse a single function with any type belonging to the `Functor` type class. `[Int]`, `Maybe Int`, and `IO Int` can all use the same core functions. Let's see if you got this.

Q27.1 When we introduced parameterized types in lesson 15, you used a minimal type `Box` as the example:

```
data Box a = Box a deriving Show
```

Implement the `Functor` type class for `Box`. Then implement `morePresents`, which changes a box from type `Box a` to one of type `Box [a]`, which has *n* copies of the original value in the box in a list. Make sure to use `fmap` to implement this.

QC27.2 Now suppose you have a simple box like this:

```
myBox :: Box Int
myBox = Box 1
```

Use `fmap` to put the value in your `Box` in another Box. Then define a function `unwrap` that takes a value out of a box, and use `fmap` on that function to get your original box. Here's how your code should work in GHCi:

```
GHCi> wrapped = fmap ? myBox
GHCi> wrapped
Box (Box 1)
GHCi> fmap unwrap wrapped
Box 1
```

Q27.3 Write a command-line interface for `partsDB` that lets the user look up the cost of an item, given an ID. Use the `Maybe` type to handle the case of the user entering missing input.

SOFTWARE DEVELOPMENT

GET PROGRAMMING WITH HASKELL

Will Kurt

Programming languages often differ only around the edges—a few keywords, libraries, or platform choices. Haskell gives you an entirely new point of view. To the software pioneer Alan Kay, a change in perspective can be worth 80 IQ points and Haskellers agree on the dramatic benefits of thinking the Haskell way—thinking functionally, with type safety, mathematical certainty, and more. In this hands-on book, that's exactly what you'll learn to do.

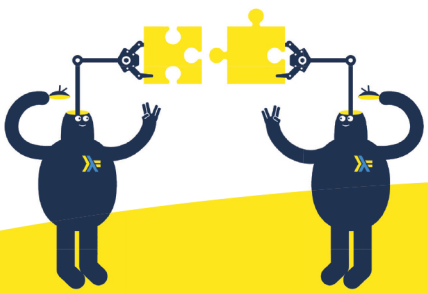
Get Programming with Haskell leads you through short lessons, examples, and exercises designed to make Haskell your own. It has crystal-clear illustrations and guided practice. You will write and test dozens of interesting programs and dive into custom Haskell modules. You will gain a new perspective on programming plus the practical ability to use Haskell in the everyday world. (The 80 IQ points: not guaranteed.)

WHAT'S INSIDE

- **Thinking in Haskell**
- **Functional programming basics**
- **Programming in types**
- **Real-world applications for Haskell**

Written for readers who know one or more programming languages.

Will Kurt *currently works as a data scientist. He writes a blog at* ***www.countbayesie.com****, explaining data science to normal people.*



To download their free eBook *in PDF, ePub, and Kindle formats, owners of this book should visit* www.manning.com/books/get-programming-with-haskell

MANNING

US $44.99 | Can $59.99 [Including eBOOK]

www.itbook.store/books/9781617293764

FREE EBOOK
See first page

"An approachable and thorough introduction to Haskell and functional programming. This book will change the way you think about programming for good."
– MAKARAND DESHPANDE
SAS R&D

"I've been trying to crack the tough nut that is Haskell for a while; I tried other books, but this was the first one that actually allowed me to understand how to use Haskell. I love how the author mixes theory with a lot of practical exercises."
– VICTOR TATAI, FITBIT

"More than a beginner's book. Full of insightful examples that make your Haskell thinking click."
– CARLOS AYA, COZERO

"I thought Haskell was hard to learn. With this book, honestly, it isn't."
– MIKKEL ARENTOFT
DANSKE BANK

"A gentle yet definitive introduction to Haskell."
– NIKITA DYUMIN
APPLIEDTECH

ISBN-13 978-1-61729-376-4
ISBN-10 1-61729-376-8
54499
9 781617 293764