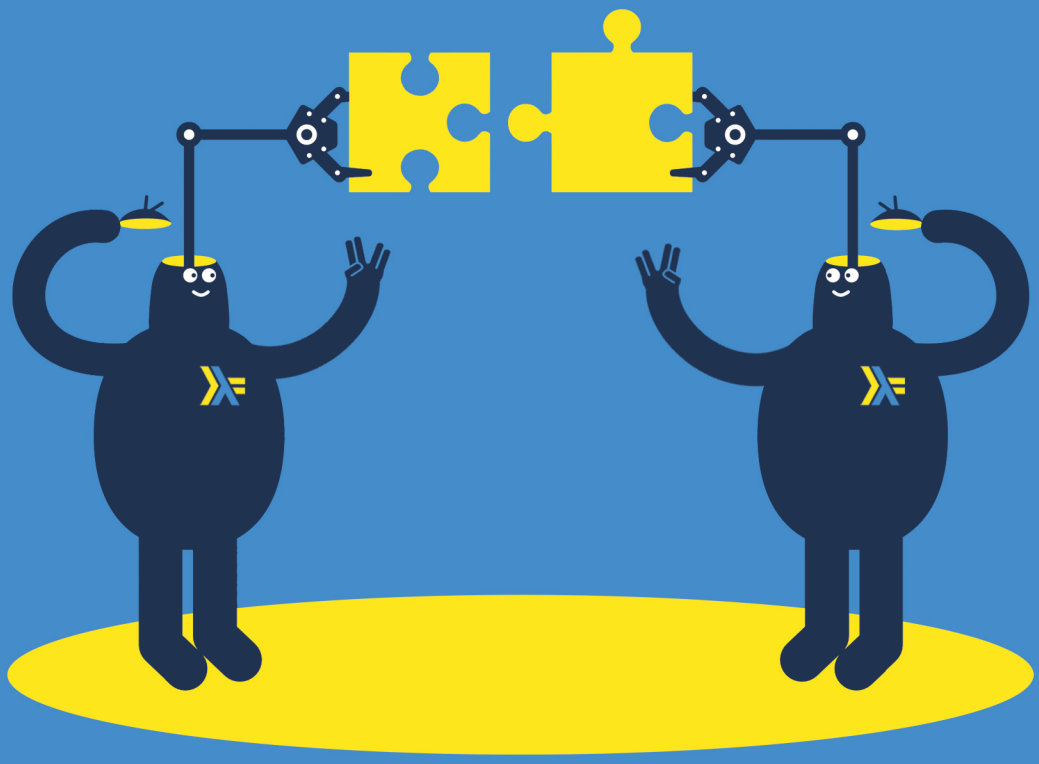
GET PROGRAMMING WITH HASKELL



Will Kurt

MANNING

Get Programming with Haskell

by Will Kurt

Lesson 5

Contents

Unit 6

ORGANIZING CODE AND BUILDING PROJECTS

Unit 7

PRACTICAL HASKELL

LESSON 5

CLOSURES AND PARTIAL APPLICATION

After reading lesson 5, you'll be able to

- Capture values in a lambda expression
- Use closures to create new functions
- Simplify this process with partial application

In this lesson, you'll learn the final key element of functional programming: closures. *Closures* are the logical consequence of having lambda functions and first-class functions. By combining these lambda functions and first-class functions to create closures, you can dynamically create functions. This turns out to be an incredibly powerful abstraction, though the one that takes the most getting used to. Haskell makes closures much easier to work with by allowing for partial application. By the end of the lesson, you'll see how partial application makes otherwise confusing closures much easier to work with.

Consider this In the preceding lesson, you learned how to pass in programming logic to other functions because of first-class functions. For example, you might have a `getPrice` function that takes a URL and a website-specific price-extraction function:

```
getPrice amazonExtractor url
```

Although this is useful, what happens if you need to extract items from 1,000 URLs, but all using `amazonExtractor`? Is there a way to capture this argument on the fly so you have to pass in only the `url` parameter for future calls?

5.1 Closures—creating functions with functions

In lesson 4, you defined a function named `ifEven` (listing 4.3). By using a function as an argument to `ifEven`, you were able to abstract out a pattern of computation. You then created the functions `ifEvenInc`, `ifEvenDouble`, and `ifEvenSquare`.

Listing 5.1 `ifEvenInc, ifEvenDouble, ifEvenSquare`

```
ifEvenInc n = ifEven inc n
ifEvenDouble n = ifEven double n
ifEvenSquare n = ifEven square n
```

Using functions as arguments helped to clean up your code. But you'll notice you're still repeating a programming pattern! Each of these definitions is identical except for the function you're passing to `ifEven`. What you want is a function that builds `ifEvenX` functions. To solve this, you can build a new function that returns functions, called `genIfEven`, as shown in figure 5.1.

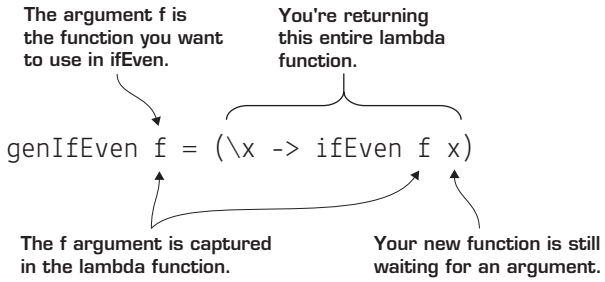


Figure 5.1 The `genIfEven` function lets you build `ifEvenX` functions simply.

Now you're passing in a function and returning a lambda function. The function `f` that you passed in is captured inside the lambda function! When you capture a value inside a lambda function, this is referred to as a *closure*.

Even in this small example, it can be difficult to understand exactly what's happening. To see this better, let's see how to create your `ifEvenInc` function by using `genIfEven`, as shown in figure 5.2.

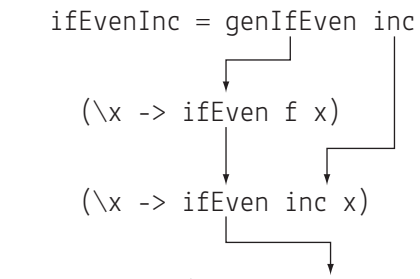


Figure 5.2 `ifEvenInc` with closure

Now let's move on to a real-world example of using closures to help build URLs to use with an API.

Quick check 5.1 Write a function `genIfXEven` that creates a closure with `x` and returns a new function that allows the user to pass in a function to apply to `x` if `x` is even.

5.2 Example: Generating URLs for an API

One of the most common ways to get data is to make calls to a RESTful API by using an HTTP request. The simplest type of request is a `GET` request, in which all of the parameters you need to send to another server are encoded in the URL. In this example, the data you need for each request is as follows:

- The hostname
- The name of the resource you're requesting
- The ID of the resource
- Your API key

Figure 5.3 shows an example URL.

QC 5.1 answer

```
ifEven f x = if even x
             then f x
             else x

genIfXEven x = (\f -> ifEven f x)
```

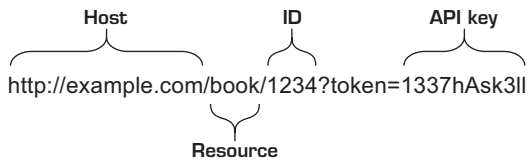


Figure 5.3 Parts of a URL

Building a URL from these parts is straightforward. Here's your basic `getRequestURL` builder.

Listing 5.2 `getRequestUrl`

```
getRequestURL host apiKey resource id = host ++
                                        "/" ++
                                        resource ++
                                        "/" ++
                                        id ++
                                        "?token=" ++
                                        apiKey
```

One thing that might strike you as odd about this function is that the order of your arguments isn't the same as the order you use them or that they appear in the URL itself. *Anytime you might want to use a closure (which in Haskell is pretty much anytime), you want to order your arguments from most to least general*. In this case, each host can have multiple API keys, each API key is going to use different resources, and each resource is going to have many IDs associated with it. The same is true when you define `ifEven`; the function you pass will work with a huge range of inputs, so it's more general and should appear first in the argument list.

Now that you have the basic request-generating function down, you can see how it works:

```
GHCi> getRequestURL "http://example.com" "1337hAsk3ll" "book" "1234"
"http://example.com/book/1234?token=1337hAsk3ll"
```

Great! This is a nice, general solution, and because your team as a whole will be querying many hosts, it's important not to be too specific. Nearly every programmer on the team will be focusing on data from just a few hosts. It seems silly, not to mention error-prone, to have programmers manually type in http://example.com every time they need to make a request. What you need is a function that everyone can use to generate a request URL builder just for them. The answer to this is a closure. Your generator will look like figure 5.4.

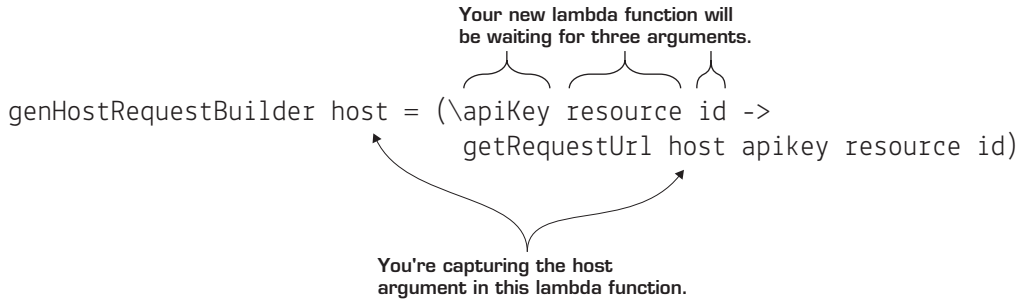


Figure 5.4 Capturing the host value in a closure

Listing 5.3 `exampleUrlBuilder` v.1

```
exampleUrlBuilder = genHostRequestBuilder "http://example.com"
```

When you pass the value `example.com`, you create a new, unnamed function that captures the host and needs only the three remaining arguments. When you define `exampleUrlBuilder`, you give a name to the anonymous function. Anytime you have a new URL that you want to make requests to, you now have an easy way to create a custom function for this. Load this function into GHCi and see how it simplifies your code:

```
GHCi> exampleUrlBuilder "1337hAsk3ll" "book" "1234"
"http://example.com/book/1234?token=1337hAsk3ll"
```

It's clear you run into the same problem again when you look at `apiKey`. Passing your API key in each time you call `exampleUrlBuilder` is still tedious because you'll likely be using only one or two API keys. Of course, you can use another closure to fix this! This time, you'll have to pass both your `exampleUrlBuilder` function and your `apiKey` to your generator.

Listing 5.4 `genApiRequestBuilder`

```
genApiRequestBuilder hostBuilder apiKey = (\resource id ->
                                            hostBuilder apiKey resource id)
```

What's interesting here is that you're combining both functions as arguments and functions as return values. Inside your closure is a copy of the specific function that you're going to need, as well as the API key you need to capture. Finally, you can build a function that makes creating a request URL much easier.

Listing 5.5 `myExampleUrlBuilder` v.1

```
myExampleUrlBuilder = genApiRequestBuilder exampleUrlBuilder "1337hAsk3ll"
```

And you can use this to quickly create URLs for different resource/ID combos:

```
GHCi> myExampleUrlBuilder "book" "1234"
"http://example.com/book/1234?token=1337hAsk3ll"
```

Quick check 5.2 Write a version of `genApiRequestBuilder` that also takes the resource as an argument.

5.2.1 Partial application: making closures simple

Closures are both powerful and useful. But the use of a lambda function to create the closure makes reading and reasoning about them more difficult than it should be. Additionally, all the closures you've written so far follow a nearly identical pattern: provide some of the parameters that a function takes and create a new function awaiting the rest. Suppose you have a function `add4` that takes four variables and adds them:

```
add4 a b c d = a + b + c + d
```

Now you want to create a function `addXto3`, which takes an argument `x` and then returns a closure awaiting the remaining three arguments:

```
addXto3 x = (\b c d ->
              add4 x b c d)
```

The explicit lambda makes it relatively hard to reason about what's happening. What if you want to make an `addXYto2`?

```
addXYto2 x y = (\c d ->
                 add4 x y c d)
```

With four arguments to manage visually, even this trivial function isn't easy to understand. Lambda functions are powerful and useful, but can definitely clutter up otherwise neat function definitions.

QC 5.2 answer

```
genApiRequestBuilder hostBuilder apiKey resource = (\id ->
                                                     hostBuilder apiKey
                                                     resource id)
```

Haskell has an interesting feature that addresses this problem. What happens if you call `add4` with fewer than four arguments? This answer seems obvious: it should throw an error. This *isn't* what Haskell does. You can define a `mystery` value in GHCi by using `Add4` and one argument:

```
GHCi> mystery = add4 3
```

If you run this code, you'll find that it doesn't cause an error. Haskell has created a brand new function for you:

```
GHCi> mystery 2 3 4
12
GHCi> mystery 5 6 7
21
```

This `mystery` function adds 3 to the three remaining arguments you pass to it. When you call any function with fewer than the required number of parameters in Haskell, you get a new function that's waiting for the remaining parameters. This language feature is called *partial application*. The mystery function is the same thing as if you wrote `addXto3` and then passed in the argument `3` to it. Not only has partial application saved you from using a lambda function, but you don't even need to define the awkwardly named `addXto3`! You can also easily re-create the behavior of `addXYto2`:

```
GHCi> anotherMystery = add4 2 3
GHCi> anotherMystery 1 2
8
GHCi> anotherMystery 4 5
14
```

If you find using closures confusing so far, you're in luck! Thanks to partial application, you rarely have to write or think explicitly about closures in Haskell. All of the work of `genHostRequestBuilder` and `genApiRequestBuilder` is built in and can be replaced by leaving out the arguments you don't need.

Listing 5.6 `exampleUrlBuilder` v.2 and `myExampleUrlBuilder` v.2

```
exampleUrlBuilder = getRequestUrl "http://example.com"
myExampleUrlBuilder = exampleUrlBuilder "1337hAsk3ll"
```

In some cases in Haskell, you'll still want to use lambda functions to create a closure, but using partial application is far more common. Figure 5.5 shows the process of partial application.

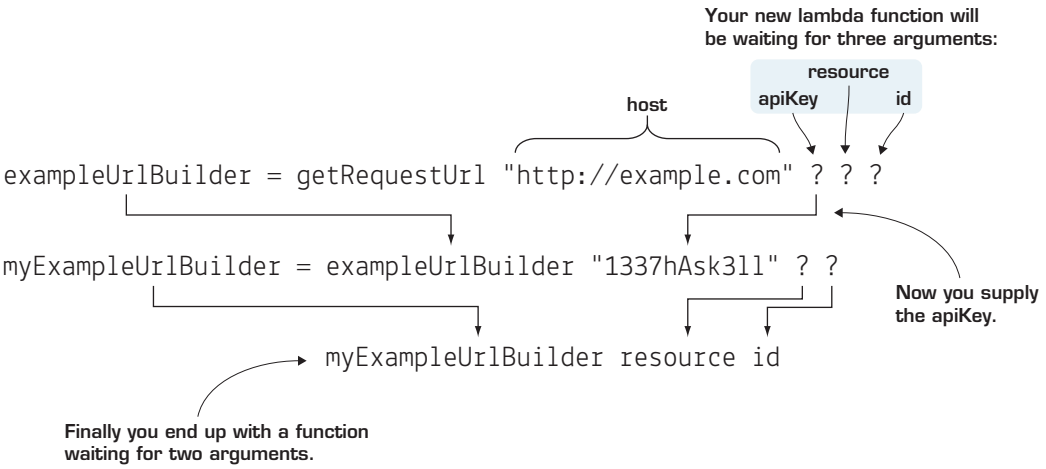


Figure 5.5 Visualizing partial application

Quick check 5.3 Make a builder function that's specifically for http://example.com, the 1337hAsk3ll API key, and the book resource. That's a function that requires only the ID of a specific book and then generates the full URL.

5.3 Putting it all together

Partial application is also the reason we created the rule that arguments should be ordered from most to least general. When you use partial application, the arguments are applied first to last. You violated this rule when you defined your `addressLetter` function in lesson 4 (listing 4.6):

```
addressLetter name location = locationFunction name
  where locationFunction = getLocationFunction location
```

In `addressLetter`, the `name` argument comes before the `location` argument. It makes much more sense that you'd want to create a function `addressLetterNY` that's waiting for a name,

QC 5.3 answer

```
exampleBuilder = getRequestUrl "http://example.com" "1337hAsk3ll" "books"
```

rather than an `addressLetterBobSmith` that will write letters to all the Bob Smiths of the world. Rather than rewriting your function, which might not always be possible if you're using functions from another library, you can fix this by creating a partial-application-friendly version, as follows.

Listing 5.7 `addressLetterV2`

```
addressLetterV2 location name = addressLetter name location
```

This is a fine solution for the one-time case of fixing your `addressLetter` function. What if you inherited a code base in which many library functions had this same error in the case of two arguments? It'd be nice to find a general solution to this problem rather than individually writing out each case. Combining all the things you've learned so far, you can do this in a simple function. You want to make a function called `flipBinaryArgs` that will take a function, flip the order of its arguments, and then return it otherwise untouched. To do this, you need a lambda function, first-class functions, and a closure. You can put all these together in a single line of Haskell, as shown in figure 5.6.

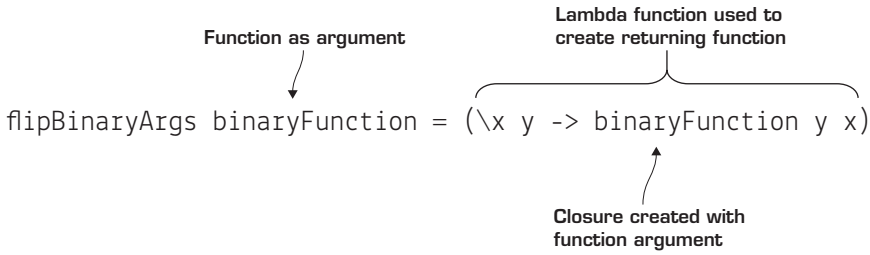


Figure 5.6 The `flipBinaryArgs` function

Now you can rewrite `addressLetterV2` by using `flipBinaryArgs`, and then create an `addressLetterNY`:

```
addressLetterV2 = flipBinaryArgs addressLetter
addressLetterNY = addressLetterV2 "ny"
```

And you can test this out in GHCi:

```
GHCi> addressLetterNY ("Bob","Smith")
Bob Smith: PO Box 789 - New York, NY, 10013
```

Your `flipBinaryArgs` function is useful for more than fixing code that didn't follow our generalization guidelines. Plenty of binary functions have a natural order, such as

division. A useful trick in Haskell is that any infix operator (such as +, /, -, *) can be used as a prefix function by putting parentheses around it:

```
GHCi> 2 + 3
5
GHCi> (+) 2 3
5
GHCi> 10 / 2
5.0
GHCi> (/) 10 2
5.0
```

In division and subtraction, the order of arguments is important. Despite there being a natural order for the arguments, it's easy to understand that you might want to create a closure around the second argument. In these cases, you can use `flipBinaryArgs` to help you. Because `flipBinaryArgs` is such a useful function, there's an existing function named `flip` that behaves the same.

Quick check 5.4 Use `flip` and partial application to create a function called `subtract2` that removes 2 from whatever number is passed in to it.

Summary

In this lesson, our objective was to teach the important idea of a closure in functional programming. With lambda functions, first-class functions, and closures, you have all you need to perform functional programming. Closures combine lambda functions and first-class functions to give you amazing power. With closures, you can easily create new functions on the fly. You also learned how partial application makes working with closures much easier. After you're used to using partial application, you may sometimes forget you're working with closures at all! Let's see if you got this.

Q5.1 Now that you know about partial application, you no longer need to use `genIfEvenX`. Redefine `ifEvenInc`, `ifEvenDouble`, and `ifEvenSquare` by using `ifEven` and partial application.

QC 5.4 answer

```
subtract2 = flip (-) 2
```

Q5.2 Even if Haskell didn't have partial application, you could hack together some approximations. Following a similar pattern to `flipBinaryArgs` (figure 5.6), write a function `binaryPartialApplication` that takes a binary function and one argument and returns a new function waiting for the missing argument.

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.

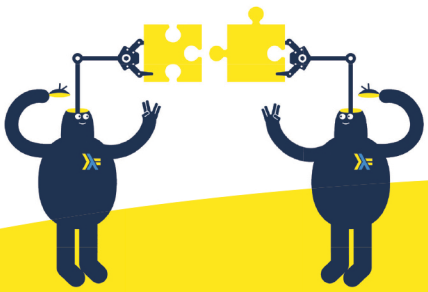
Get Programming with Haskell leads you through short lessons, examples, and exercises designed to make Haskell your own. It has crystal-clear illustrations and guided practice. You will write and test dozens of interesting programs and dive into custom Haskell modules. You will gain a new perspective 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]

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