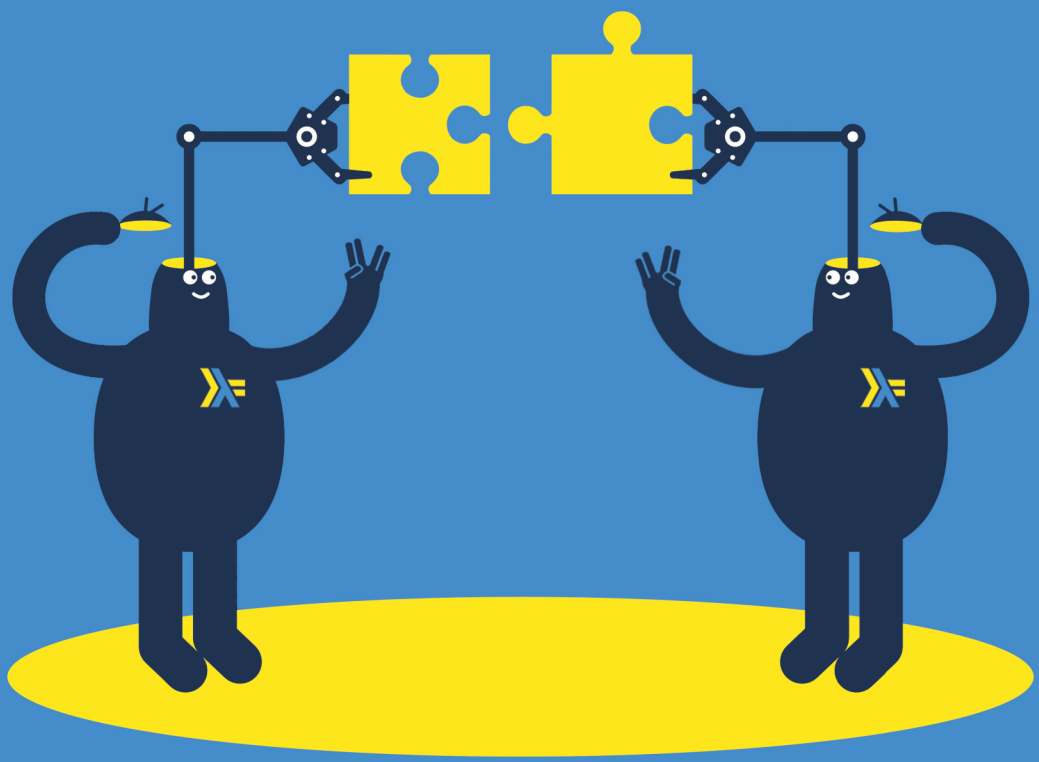
GET PROGRAMMING WITH HASKELL



Will Kurt

MANNING

Get Programming with Haskell

by Will Kurt

Lesson 38

Contents

Unit 6

ORGANIZING CODE AND BUILDING PROJECTS

Unit 7

PRACTICAL HASKELL

LESSON 38

ERRORS IN HASKELL AND THE EITHER TYPE

After reading lesson 38, you'll be able to

- Throw errors by using the `error` function
- Understand the dangers of throwing errors
- Use `Maybe` as a method for handling errors
- Handle more sophisticated errors with the `Either` type

Most of what makes Haskell so powerful is based on the language being safe, predictable, and reliable. Although Haskell reduces or eliminates many problems, errors are an unavoidable part of real-world programming. In this lesson, you'll learn how to think about handling errors in Haskell. The traditional approach of throwing an exception is frowned upon in Haskell, as this makes it easy to have runtime errors the compiler can't catch. Although Haskell does allow you to throw errors, there are better ways to solve many problems that come up in your programs. You've already spent a lot of time with one of these methods: using the `Maybe` type. The trouble with `Maybe` is that you don't have a lot of options for communicating what went wrong. Haskell provides a more powerful type, `Either`, that lets you use any value you'd like to provide information about an error.

In this lesson, you'll use the `error` function, `Maybe` type, and `Either` type in Haskell to resolve exceptions in your programs. You'll start by exploring the `head` function. Though

`head` is one of the first functions you learned, its implementation has a major issue: it's easy to use `head` and create runtime errors that can't be captured by Haskell's type system. You'll look at several alternative ways to handle the case where `head` fails. This problem can be better solved by using the familiar `Maybe` type, and you can give more informative errors by using a new type you'll learn about, called `Either`. You'll conclude by building a simple command-line tool that checks whether a number is prime. You'll use the `Either` type and your own error data type to represent errors and display them to the user.

Consider this You have a list representing employee ID numbers. Employee IDs can't be larger than 10000 or less than 0. You have an `idInUse` function that checks whether a specific ID is being used. How can you write a function that lets a programmer using `idInUse` distinguish between a user that isn't in the database and a value that's outside the range of valid employee IDs?

38.1 Head, partial functions, and errors

One of the first functions you were introduced to was `head`. The `head` function gives you the first element in the list, if there is one. The trouble with `head` is what to do when there's no first element in the list (an empty list). See figure 38.1.

Initially, `head` seems like an incredibly useful function. Many recursive functions you write in Haskell use lists, and accessing the first item in a list is a common requirement.

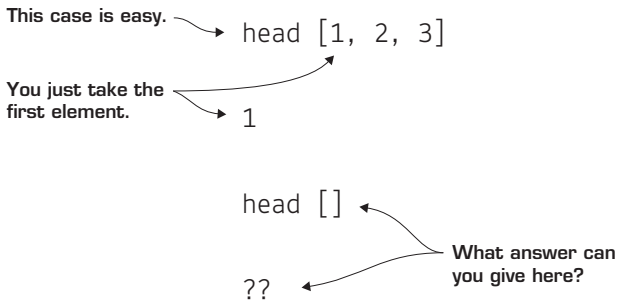


Figure 38.1 How can you solve the case of calling `head` on an empty list?

But `head` has one big issue. When you call `head` on an empty list, you get an error:

```
GHCi> head [1]
1
GHCi> head []
*** Exception: Prelude.head: empty list
```

In most programming languages, throwing an exception like this is common practice. In Haskell, this is a big problem, because throwing an exception makes your code unpredictable. One of the key benefits of using Haskell is that your programs are safer and more predictable. But nothing about the `head` function, or its type signature, gives you any indication that it could suddenly blow up:

```
head :: [a] -> a
```

By this point in the book, you've seen firsthand that if a Haskell program compiles, it likely runs as expected. But `head` violates this rule by making it easy to write code that compiles but then causes an error at runtime.

For example, suppose you naively implement a recursive `myTake` function using `head` and `tail`.

Listing 38.1 A function that easily causes an error when used but compiles fine

```
myTake :: Int -> [a] -> [a]
myTake 0 _ = []
myTake n xs = head xs : myTake (n-1) (tail xs)
```

Let's compile this code, only this time you'll set a compiler flag to warn of any potential problems with the code. You can do this by using the `-Wall` flag. This can be done in stack by adding `-Wall` to the `ghc-options` value in the executable section of the .cabal file. As a refresher from lesson 35, open the headaches.cabal file in the projects root directory, find the executable section of the .cabal file, and append `-Wall` to the list of `ghc-options` as shown here:

The -Wall argument sets all warnings to be checked when the program is compiled.

```
executable headaches-exe
  hs-source-dirs:      app
  main-is:             Main.hs
  ghc-options:         -threaded -rtsopts -with-rtsopts=-N -Wall
  build-depends:       base
                     , headaches
  default-language:    Haskell2010
```

After you change your file, you need to restart GHCi (which will automatically rebuild your project). Now if you build your project, you'll get no complaints from the compiler. But it's trivial to see that this code produces an error:

```
GHCi> myTake 2 [1,2,3] :: [Int]
[1,2]
GHCi> myTake 4 [1,2,3] :: [Int]
[1,2,3,*** Exception: Prelude.head: empty list
```

Imagine that this code is running and processing requests from a user. This kind of failure would be frustrating, especially given that you're using Haskell.

To understand why `head` is so dangerous, let's compare this to the exact same version using pattern matching.

Listing 38.2 An identical function to `myTake`, which throws a compiler warning

```
myTakePM :: Int -> [a] -> [a]
myTakePM 0 _ = []
myTakePM n (x:xs) = x : myTakePM (n-1) xs
```

This code is identical in behavior to `myTake`, but when you compile with `-Wall`, you get a helpful error:

```
Pattern match(es) are non-exhaustive
    In an equation for 'myTakePM':
        Patterns not matched: p [] where p is not one of {0}
```

This tells you that your function doesn't have a pattern for the empty list! Even though this is identical to the code using `head`, GHC can warn you about this.

> **NOTE** If you don't want to miss warnings on large projects, you can compile with `-error`, which causes an error anytime a warning is found.

Quick check 38.1 Which of the following is the missing pattern that would fix `myTakePM`?

```
myTakePM _0 [] = []

myTakePM _ [] = []

myTakePM 0 (x:xs) = []
```

38.1.1 Head and partial functions

The `head` function is an example of a *partial function*. In lesson 2, you learned that every function must take an argument and return a result. Partial functions don't violate this rule, but they have one significant failing. They aren't defined on all inputs. The `head` function is undefined on the empty list.

Nearly all errors in software are the result of partial functions. Your program receives input you don't expect, and the program has no way of dealing with it. Throwing an error is an obvious solution to this problem. Throwing errors in Haskell is simple: you use the `error` function. Here's `myHead` with an error.

Listing 38.3 `myHead`, an example of throwing an error

```
myHead :: [a] -> a
myHead [] = error "empty list"
myHead (x:_) = x
```

Throws an error whenever myHead matches an empty list

In Haskell, throwing errors is considered bad practice. This is because, as you saw with `myTake`, it's easy to introduce bugs into code that the compiler can't check. In practice, you should *never* use `head`, and instead use pattern matching. If you replace any instance of using `head` and `tail` in your code with pattern matching, the compiler can warn you of errors.

The real question is, what do you do about partial functions in general? Ideally, you want a way to transform your partial function into one that works on all values. Another common partial function is `(/)`, which is undefined for 0. But Haskell avoids throwing an error in this case by providing a different solution:

```
GHCi>  2 / 0
Infinity
```

QC 38.1 answer You need to add the following pattern:

```
myTakePM _ [] = []
```

This is a nice solution to the problem of dividing by zero, but solutions like this exist for only a few specific cases. What you want is a way to use types to capture when errors might happen. Your compiler can help in writing more error-resistant code.

Quick check 38.2 The following are all partial functions included in `Prelude`. For what inputs do they fail?

- `maximum`
- `succ`
- `sum`

38.2 Handling partial functions with Maybe

It turns out you've already seen one of the most useful ways to handle partial functions: `Maybe`. In many of the examples of `Maybe` that you've used, there would be a `Null` value in other languages. But `Maybe` is a reasonable way to transform any partial function into a complete function. Here's your code for `maybeHead`.

Listing 38.4 Using `Maybe` to make `head` a complete function

```
maybeHead :: [a] -> Maybe a
maybeHead [] = Nothing
maybeHead (x:_) = Just x
```

With `maybeHead`, you can safely take the `head` of a list:

```
GHCi> maybeHead [1]
Just 1
GHCi> maybeHead []
Nothing
```

In unit 5, you learned that `Maybe` is an instance of `Monad` (and therefore `Functor` and `Applicative`), which allows you to perform computation on values in a `Maybe` context. Recall that the

QC 38.2 answer

- `maximum`—Fails on the empty list
- `succ`—Fails on `maxBound` for the type
- `sum`—Fails on infinite lists

`Functor` type class allows you to use `<$>` to apply a function to a `Maybe` value. Here's an example of using the `maybeHead` function, as well as using `<$>` to operate on the values it produces:

```
GHCi> (+2) <$> maybeHead [1]
Just 3
GHCi> (+2) <$> maybeHead []
Nothing
```

The `Applicative` type class provides the `<*>` operator, so you can chain together functions in a context, most commonly used for multi-argument functions. Here's how to use `<$>` with `<*>` to cons a result from `maybeHead` with a `Just []`:

```
GHCi> (:) <$> maybeHead [1,2,3] <*> Just []
Just [1]
GHCi> (:) <$> maybeHead [] <*> Just []
Nothing
```

You can combine `maybeHead` with `<$>` and `<*>` to write a new, safer version of `myTake`.

Listing 38.5 A safer version of `myTake` using `maybeHead` instead of `head`

```
myTakeSafer :: Int -> Maybe [a] -> Maybe [a]
myTakeSafer 0 _ = Just []
myTakeSafer n (Just xs) = (:) <$> maybeHead xs
                              <*> myTakeSafer (n-1) (Just (tail xs))
```

In GHCi, you can see that the `myTakeSafer` function works just fine with error-causing inputs:

```
GHCi> myTakeSafer 3 (Just [1,2,3])
Just [1,2,3]
GHCi> myTakeSafer 6 (Just [1,2,3])
Nothing
```

As you can see, `myTakeSafer` works as you'd expect (though differently than take, which would return the full list). Note that the reason you named it `safer`, not `safe`, is that, unfortunately, *tail is also a partial function*.

38.3 Introducing the Either type

We've spent a lot of time in this book talking about the power of `Maybe`, but it does have one major limitation. As you write more sophisticated programs, the `Nothing` result becomes harder to interpret. Recall that in our unit 6 capstone you had an `isPrime` function. Here's a simplified version of `isPrime`:

```
primes :: [Int]
primes = [2,3,5,7]

maxN :: Int
maxN = 10

isPrime :: Int -> Maybe Bool
isPrime n
   | n < 2 = Nothing
   | n > maxN = Nothing
   | otherwise = Just (n `elem` primes)
```

The list of primes you're using to determine whether a number is prime

The largest value you'll check for primality

If the number is less than 2, you don't consider checking it.

If the number is greater than your max, you can't know whether it's prime.

If the number is a valid candidate, check whether it's prime.

You made this function of type `Int -> Maybe Bool` because you wanted to handle your edge cases. The key issue is that you want a `False` value for `isPrime` to mean that a number is composite. But there are two problems. Numbers such as 0 and 1 are neither composite nor prime. Additionally, the `isPrime` function limits how large a number can be, and you don't want to return `False` just because a value is too expensive to compute.

Now imagine you're using `isPrime` in your own software. When you call `isPrime 9997`, you get `Nothing` as a result. What in the world does this mean? You'd have to look up the documentation (hoping there is any) to find out. The nice thing about errors is that you get an error message. Although `Maybe` does give you lots of safety, unless `Nothing` has an obvious interpretation, as in the case of `Null` values, it's not useful. Fortunately, Haskell has another type, similar to `Maybe`, that allows you to create much more expressive errors while remaining safe.

Quick check 38.3 Suppose you have this list:

```
oddList :: [Maybe Int]
oddList = [Nothing]
```

The type you'll be looking at is called `Either`. Though only a bit more complicated than `Maybe`, its definition can be confusing. Here's the definition of `Either`:

```
data Either a b = Left a | Right b
```

`Either` has two confusingly named data constructors: `Left` and `Right`. For handling errors, you can consider the `Left` constructor as the case of having an error, and the `Right` constructor for when things go as planned. A more user-friendly, but less general way to define `Either` is as follows:

```
data Either a b = Fail a | Correct b
```

In practice, the `Right` constructor works exactly like `Just` for `Maybe`. The key difference between the two is that `Left` allows you to have more information than `Nothing`. Also notice that `Either` takes two type parameters. This allows you to have a type for sending error messages and a type for your actual data. To demonstrate, here's an example of making a safer `head` function with `Either`.

Listing 38.6 A safer version of `head` written using `Either`

```
eitherHead :: [a] -> Either String a
eitherHead [] = Left "There is no head because the list is empty"
eitherHead (x:xs) = Right x
```

Notice that the `Left` constructor takes a `String`, whereas the `Right` constructor returns the value from the first item in your list. Here are some example lists you can test on:

```
intExample :: [Int]
intExample = [1,2,3]

intExampleEmpty :: [Int]
intExampleEmpty = []
```

QC 38.3 answer

```
Maybe Int
```

```
charExample :: [Char]
charExample = "cat"

charExampleEmpty :: [Char]
charExampleEmpty = ""
```

In GHCi, you can see how `Either` works, as well as the types you get back:

```
GHCi> eitherHead intExample
Right 1
GHCi> eitherHead intExampleEmpty
Left "There is no head because the list is empty"
GHCi> eitherHead charExample
Right 'c'
GHCi> eitherHead charExampleEmpty
Left "There is no head because the list is empty"
```

The `Either` type is also a member of `Monad` (and thus `Functor` and `Applicative` as well). Here's a simple example of using `<$>` to increment the `head` of your `intExample`:

```
GHCi> (+ 1) <$> (eitherHead intExample)
Right 2
GHCi> (+ 1) <$> (eitherHead intExampleEmpty)
Left "There is no head because the list is empty"
```

The `Either` type combines the safety of `Maybe` with the clarity that error messages provide.

Quick check 38.4 Use `<$>` and `<*>` to add the first and second numbers in `intExample` by using `eitherHead`.

38.3.1 Building a prime check with Either

To demonstrate working with `Either`, let's see how to build a basic command-line tool to check whether a number is prime. You'll keep your `isPrime` function minimal, focusing on using the `Either` type. You'll begin by using a `String` for your error message. Then

QC 38.4 answer

```
(+) <$> eitherHead intExample <*> eitherHead (tail intExample)
```

you'll take advantage of the fact that `Either` lets you use any type you want to, allowing you to create your own error types.

The nice thing about `Either` is you don't have to stick to a single error message. You can have as many as you'd like. Your improved `isPrime` function will let you know whether a value isn't a valid candidate for primality checking, or whether the number is too large.

Listing 38.7 `isPrime` refactors to use multiple messages when a number is invalid

```
isPrime :: Int -> Either String Bool
isPrime n
   | n < 2 = Left "Numbers less than 2 are not candidates for primes"
   | n > maxN = Left "Value exceeds limits of prime checker"
   | otherwise = Right (n `elem` primes)
```

Here are a few tests of this function in GHCi:

```
GHCi> isPrime 5
Right True
GHCi> isPrime 6
Right False
GHCi> isPrime 100
Left "Value exceeds limits of prime checker"
GHCi> isPrime (-29)
Left "Numbers less than 2 are not candidates for primes"
```

So far, you haven't taken advantage of `Either` being able to take two types; you've exclusively used `String` for the `Left` constructor. In most programming languages, you can represent errors by using a class. This makes it easier to model specific types of errors. `Either` allows you to do this as well. You'll start by making your errors into a type of their own.

Listing 38.8 The `PrimeError` types for representing your errors as types

```
data PrimeError = TooLarge | InvalidValue
```

Now you can make this an instance of `Show` so you can easily print out these errors.

Listing 38.9 Making `PrimeError` an instance of `Show`

```
instance Show PrimeError where
   show TooLarge = "Value exceed max bound"
   show InvalidValue = "Value is not a valid candidate for prime checking"
```

With your new `PrimeError` type, you can refactor your `isPrime` function to show off these errors.

Listing 38.10 Refactoring `isPrime` to use `PrimeError`

```
isPrime :: Int -> Either PrimeError Bool
isPrime n
   | n < 2 = Left InvalidValue
   | n > maxN = Left TooLarge
   | otherwise = Right (n `elem` primes)
```

This makes your code much more readable. Additionally, you now have an easily reusable data type that will work with your errors. Here are some examples of your new function in GHCi:

```
GHCi> isPrime 99
Left Value exceed max bound
GHCi> isPrime 0:
Left Value is not a valid candidate for prime checking
```

Next you'll create a `displayResult` function that will convert your `Either` response into a `String`.

Listing 38.11 Translating your `isPrime` result to be human-readable

```
displayResult :: Either PrimeError Bool -> String
displayResult (Right True) = "It's prime"
displayResult (Right False) = "It's composite"
displayResult (Left primeError) = show primeError
```

Finally, you can put together a simple `main IO` action that reads as follows.

Listing 38.12 The `main` to check for primes from user input

```
main :: IO ()
main = do
  print "Enter a number to test for primality:"
  n <- read <$> getLine
  let result = isPrime n
  print (displayResult result)
```

Now you can build and run your program:

```
$ stack build
$ stack exec primechecker-exe
"Enter a number to test for primality:"
6
"It's composite"

$ stack exec headaches-exe
"Enter a number to test for primality:"
5
"It's prime"

$ stack exec headaches-exe
"Enter a number to test for primality:"
213
"Value exceed max bound"

$ stack exec headaches-exe
"Enter a number to test for primality:"
0
"Value is not a valid candidate for prime checking"
```

With your `PrimeError` type, you were able to replicate more sophisticated ways of modeling errors in OOP languages. The great thing about `Either` is that because the `Left` constructor can be any type, there's no limit to how expressive you can be. If you wanted to, you could return a function!

Summary

In this lesson, our objective was to teach you how to safely handle errors in Haskell. You started by looking at the way `head` uses `error` to signal when you have an empty list with no head. Neither your type checker nor GHC's warnings let you know this is a problem. This is ultimately caused by `head` being a partial function, a function that doesn't return a result for all possible inputs. This can be solved by using a `Maybe` type. Although `Maybe` types do make your code safer, they can make errors hard to understand. Finally, you saw that the `Either` type provides the best of both worlds, allowing you to safely handle errors as well as providing detailed information about them.

Q38.1 Make a function `addStrInts` that takes two `Int`s represented as `String`s and adds them. The function would return an `Either String Int`. The `Right` constructor should return the result, provided that the two arguments can be parsed into `Int`s (use `Data.Char isDigit` to check). Return a different `Left` result for the three possible cases:

- First value can't be parsed.
- Second value can't be parsed.
- Neither value can be parsed.

Q38.2 The following are all partial functions. Use the type specified to implement a safer version of the function:

- `succ`—`Maybe`
- `tail`—`[a]` (Keep the type the same.)
- `last`—`Either` (`last` fails on empty lists and infinite lists; use an upper bound for the infinite case.)

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.

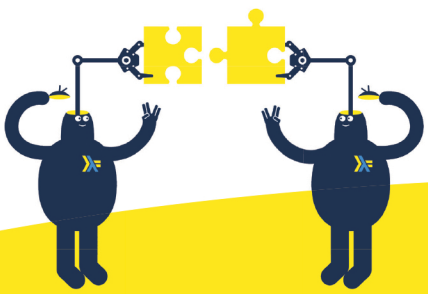
Get Programming with Haskell leads you through short lessons, examples, and exercises designed to make Haskell your own. It has crystal-clear illustrations and guided practice. You will write and test dozens of interesting programs and dive into custom Haskell modules. You will gain a new perspective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