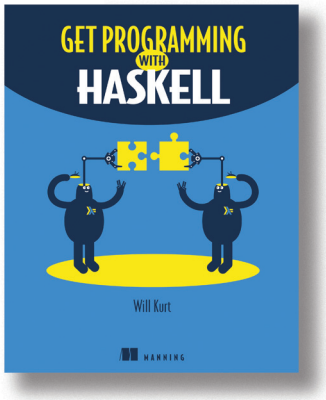
FREE

Exploring Haskell: Core Concepts

Chapters selected by Marcello Seri

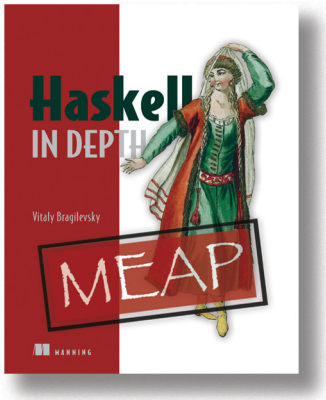
manning



Get Programming with Haskell

by Will Kurt

ISBN 9781617293764
616 pages
$44.99
March 2018



Haskell in Depth

by Vitaly Bragilevsky

ISBN 9781617295409
625 pages (estimated)
$49.99
Fall 2019

Exploring Haskell: Core Concepts

Chapters Selected by Marcello Seri

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617296772
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 24 23 22 21 20 19

contents

introduction

With the increasing size and complexity of modern software, and with the steady increase in the number of cores and parallel processes in CPU and GPU architectures, the benefits of type safety guarantees have become more and more evident.

Since 1990, when Haskell 1.0 appeared, the language has evolved steadily. From being a niche academic toy for type theorists, it has become a leading industrial-strength language that has influenced a large family of modern programming languages. Nowadays, you can find Haskell at the core of Facebook antispam infrastructure, providing an engine generate correct and type-checked hardware circuits and differentiation engines, to give you some examples.

Despite its wide adoption and renowned power, Haskell has a steep learning curve. The documentation, even if it has made leaps forward in the past few years, leaves a lot to be desired, and its deep, historical connection to category theory research can scare away even the toughest developers. If that weren't enough, Haskell is different—very different—from most other programming languages. If you persist enough to break the wall, you'll discover a language that forces you to think in different ways, approach problems from new angles, and be amazed by the power of a truly amazing type system.

This small collection of chapters, extracted from Will Kurt's *Get Programming with Haskell* and Vitaly Bragilevsky's *Haskell in Depth*, hopes to give you a glimpse into the world of Haskell. The selected chapters give you some insight into how easy it has become to get started with Haskell, walk you through some of the core concepts, and terminate with an elegant real-world example of a nontrivial CLI.

I hope this brief encounter with Haskell will invite you to keep going and deepening your knowledge into the amazing world of functional programming.

LESSON 1

Lesson 1 from *Get Programming with Haskell* by Will Kurt

GETTING STARTED WITH HASKELL

After reading lesson 1, you'll be able to

- Install tools for Haskell development
- Use GHC and GHCi
- Use tips for writing Haskell programs

1.1 Welcome to Haskell

Before you dive into learning Haskell, you need to become familiar with the basic tools you'll be using on your journey. This lesson walks you through getting started with Haskell. The lesson starts with downloading the basics to write, compile, and run Haskell programs. You'll then look at example code and start thinking about how to write code in Haskell. After this lesson, you'll be ready to dive in!

1.1.1 The Haskell Platform

The worst part of learning a new programming language is getting your development environment set up for the first time. Fortunately, and somewhat surprisingly, this isn't a problem at all with Haskell. The Haskell community has put together a single, easily installable package of useful tools referred to as the *Haskell Platform*. The Haskell Platform is the "batteries included" model of packaging a programming language.

The Haskell Platform includes the following:

- The Glasgow Haskell Compiler (GHC)
- An interactive interpreter (GHCi)
- The stack tool for managing Haskell projects
- A bunch of useful Haskell packages

The Haskell Platform can be downloaded from https://www.haskell.org/downloads#platform. From there, follow the directions for installing on your OS of choice. This book uses Haskell version 8.0.1 or higher.

1.1.2 Text editors

Now that you have the Haskell Platform installed, you're probably curious about which editor you should use. Haskell is a language that strongly encourages you to *think before you hack*. As a result, Haskell programs tend to be extremely terse. There's little that an editor can do for you, other than manage indentation and provide helpful syntax highlighting. Many Haskell developers use Emacs with `haskell-mode`. But if you're not already familiar with Emacs (or don't like to work with it), it's certainly not worth the work to learn Emacs in addition to Haskell. My recommendation is that you look for a Haskell plugin for whatever editor you use the most. A bare-bones text editor, such as Pico or Notepad++, will work just fine for this book, and most full-fledged IDEs have Haskell plugins.

1.2 The Glasgow Haskell Compiler

Haskell is a compiled language, and the Glasgow Haskell Compiler is the reason Haskell is as powerful as it is. The job of the compiler is to transform human-readable source code into machine-readable binary. At the end compilation, you're left with an executable binary file. This is different from when you run Ruby, for example, in which another program reads in your source code and interprets it on the fly (this is accomplished with an *interpreter*). The main benefit of a compiler over an interpreter is that because the compiler transforms code in advance, it can perform analysis and optimization of the code you've written. Because of some other design features of Haskell, namely its powerful type system, there's an adage that *if it compiles, it works*. Though you'll use GHC often, never take it for granted. It's an amazing piece of software in its own right.

To invoke GHC, open a terminal and type in `ghc`:

```
$ ghc
```

In this text, whenever you come across a `$` sign, it means you're typing into a command prompt. Of course, with no file to compile, GHC will complain. To get started, you'll make a simple file called hello.hs. In your text editor of choice, create a new file named hello.hs and enter the following code.

Listing 1.1 hello.hs a Hello World program

```
--hello.hs my first Haskell file!
main = do
  print "Hello World!"
```

A commented line with the name of your file

The start of your 'main' function

The main function prints out "Hello World"

At this point, don't worry too much about what's happening in any of the code in this section. Your real aim here is to learn the tools you need so that they don't get in the way while you're learning Haskell.

Now that you have a sample file, you can run GHC again, this time passing in your hello.hs file as an argument:

```
$ ghc hello.hs
[1 of 1] Compiling Main
Linking hello ...
```

If the compilation was successful, GHC will have created three files:

- hello (hello.exe on Windows)
- hello.hi
- hello.o

Starting out, the most important file is hello, which is your binary executable. Because this file is a binary executable, you can simply run the file:

```
$ ./hello
"Hello World!"
```

Notice that the default behavior of the compiled program is to execute the logic in `main`. By default, all Haskell programs you're compiling need to have a `main`, which plays a similar role to the `Main` method in Java/C++/C# or `__main__` in Python.

Like most command-line tools, GHC supports a wide range of optional flags. For example, if you want to compile hello.hs into an executable named helloworld, you can use the `-o` flag:

```
$ghc hello.hs -o helloword
[1 of 1] Compiling Main
Linking helloworld ....
```

For a more complete listing of compiler options, call `ghc --help` (no filename argument is required).

Quick check 1.1 Copy the code for hello.hs and compile your own executable named testprogram.

1.3 Interacting with Haskell—GHCi

One of the most useful tools for writing Haskell programs is GHCi, an interactive interface for GHC. Just like GHC, GHCi is started with a simple command: `ghci`. When you start GHCi, you'll be greeted with a new prompt:

```
$ ghci
GHCi>
```

This book indicates when you're using GHCi by using `GHCi>` for lines you input and a blank for lines that are output by GHCi. The first thing to learn about any program you start from the command line is how to get out of it! For GHCi, you use the `:q` command to exit:

```
$ ghci
GHCi> :q
Leaving GHCi.
```

Working with GHCi is much like working with interpreters in most interpreted programming languages such as Python and Ruby. It can be used as a simple calculator:

```
GHCi> 1 + 1
2
```

You can also write code on the fly in GHCi:

```
GHCi> x = 2 + 2
GHCi> x
4
```

QC 1.1 answer Simply copy the code to a file and then run this in the same directory as the file: `ghc hello.hs -o testprogram`

Prior to version 8 of GHCi, function and variable definitions needed to be prefaced with a `let` keyword. This is no longer necessary, but many Haskell examples on the web and in older books still include it:

```
GHCi> let f x = x + x
GHCi> f 2
4
```

The most important use of GHCi is interacting with programs that you're writing. There are two ways to load an existing file into GHCi. The first is to pass the filename as an argument to `ghci`:

```
$ ghci hello.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
```

The other is to use the `:l` (or `:load`) command in the interactive session:

```
$ ghci
GHCi> :l hello.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
```

In either of these cases, you can then call functions you've written:

```
GHCi> :l hello.hs
GHCi> main
"Hello World!"
```

Unlike compiling files in GHC, your files don't need a `main` in order to be loaded into GHCi. Anytime you load a file, you'll overwrite existing definitions of functions and variables. You can continually load your file as you work on it and make changes. Haskell is rather unique in having strong compiler support as well as a natural and easy-to-use interactive environment. If you're coming from an interpreted language such as Python, Ruby, or JavaScript, you'll feel right at home using GHCi. If you're familiar with compiled languages such as Java, C#, or C++, you'll likely be surprised that you're working with a compiled language when writing Haskell.

Quick check 1.2 Edit your Hello World script to say Hello <Name> with your name. Reload this into GHCi and test it out.

14 Writing and working with Haskell code

One of the most frustrating issues for newcomers to Haskell is that basic I/O in Haskell is a fairly advanced topic. Often when new to a language, it's a common pattern to print output along the way to make sure you understand how a program works. In Haskell, this type of ad hoc debugging is usually impossible. It's easy to get a bug in a Haskell program, along with a fairly sophisticated error, and be at an absolute loss as to how to proceed.

Compounding this problem is that Haskell's wonderful compiler is also strict about the correctness of your code. If you're used to writing a program, running it, and quickly fixing any errors you made, Haskell will frustrate you. Haskell strongly rewards taking time and thinking through problems before running programs. After you gain experience with Haskell, I'm certain that these frustrations will become some of your favorite features of the language. The flipside of being obsessed with correctness during compilation is that programs will work, and work as expected far more often than you're likely used to.

The trick to writing Haskell code with minimal frustration is to write code in little bits, and play with each bit interactively as it's written. To demonstrate this, you'll take a messy Haskell program and clean it up so it's easy to understand each piece. For this example, you'll write a command-line app that will draft *thank-you* emails to readers from authors. Here's the first, poorly written, version of the program.

QC 1.2 answer Edit your file so that it has your name:

```
main = do
  print "Hello Will!"
```

In GHCi, load your file:

```
GHCi> :l hello.hs
GHCi> main
Hello Will!
```

Listing 1.2 A messy version of first_prog.hs

```
messyMain :: IO()
messyMain = do
   print "Who is the email for?"
   recipient <- getLine
   print "What is the Title?"
   title <- getLine
   print "Who is the Author?"
   author <- getLine
   print ("Dear " ++ recipient ++ ",\n" ++
     "Thanks for buying " ++ title  ++ "\nthanks,\n" ++
     author )
```

The key issue is that this code is in one big monolithic function named `messyMain`. The advice that it's good practice to write modular code is fairly universal in software, but in Haskell it's essential for writing code that you can understand and troubleshoot. Despite being messy, this program does work. If you changed the name of `messyMain` to `main`, you could compile and run this program. But you can also load this code into GHCi as it is, assuming that you're in the same directory as your first_prog.hs:

```
$ghci
GHCi> :l first_prog.hs
[1 of 1] Compiling Main             ( first_prog.hs, interpreted )
Ok, modules loaded: Main.
```

If you get the `Ok` from GHCi, you know that your code compiled and works just fine! Notice that GHCi doesn't care if you have a `main` function. This is great, as you can still interact with files that don't have a `main`. Now you can take your code for a test drive:

```
GHCi> messyMain
"Who is the email for?"
Happy Reader
"What is the Title?"
Learn Haskell
"Who is the Author?"
Will Kurt
"Dear Happy Reader,\nThanks for buying Learn Haskell\nthanks,\nWill Kurt"
```

Everything works fine, but it'd be much easier to work with if this code was broken up a bit. Your primary goal is to create an email, but it's easy to see that the email consists of

tying together three parts: the recipient section, the body, and the signature. You'll start by pulling out these parts into their own functions. The following code is written into your first_prog.hs file. Nearly all of the functions and values defined in this book can be assumed to be written into a file you're currently working with. You'll start with just the `toPart` function:

```
toPart recipient = "Dear" ++ recipient ++ ",\n"
```

In this example, you could easily write these three functions together, but it's often worth it to work slowly and test each function as you go. To test this out, you'll load your file again in GHCi:

```
GHCi> :l "first_prog.hs"
[1 of 1] Compiling Main             ( first_prog.hs, interpreted )
Ok, modules loaded: Main.
GHCi> toPart "Happy Reader"
"DearHappy Reader,\n"
GHCi> toPart "Bob Smith"
"DearBob Smith,\n"
```

This pattern of writing code in an editor and then loading and reloading it into GHCi will be your primary means of working with code throughout the book. To avoid repetition, the `:l "first_prog.hs"` will be assumed rather than explicitly written from here on.

Now that you've loaded this into GHCi, you see there's a slight error, a missing space between *Dear* and the recipient's name. Let's see how to fix this.

Listing 1.3 Corrected `toPart` function

```
toPart recipient = "Dear " ++ recipient ++ ",\n"
```

And back to GHCi:

```
GHCi> toPart "Jane Doe"
"Dear Jane Doe,\n"
```

Everything looks good. Now to define your two other functions. This time you'll write them both at the same time. While following along, it's still a good idea to write code one function at a time, load it into GHCi, and make sure it all works before moving on.

Listing 1.4 Defining the `bodyPart` and `fromPart` functions

```
bodyPart bookTitle = "Thanks for buying " ++ bookTitle ++ ".\n"
fromPart author = "Thanks,\n"++author
```

You can test these out as well:

```
GHCi> bodyPart "Learn Haskell"
"Thanks for buying Learn Haskell.\n"
GHCi> fromPart "Will Kurt"
"Thanks,\nWill Kurt"
```

Everything is looking good! Now you need a function to tie it all together.

Listing 1.5 Defining the `createEmail` function

```
createEmail recipient bookTitle author = toPart recipient ++
                                         bodyPart bookTitle ++
                                         fromPart author
```

Notice the alignment of the three function calls. Haskell makes limited use of significant whitespace (but nothing as intense as Python). Assume that any formatting in this text is intentional; if sections of code are lined up, it's for a reason. Most editors can handle this automatically with a Haskell plugin.

With all your functions written, you can test `createEmail`:

```
GHCi> createEmail "Happy Reader" "Learn Haskell" "Will Kurt"
"Dear Happy Reader,\nThanks for buying Learn Haskell.\nThanks,\nWill Kurt"
```

Your functions each work as expected. Now you can put them all together in your `main`.

Listing 1.6 Improved first_prog.hs with a cleaned-up `main`

```
main = do
   print "Who is the email for?"
   recipient <- getLine
   print "What is the Title?"
   title <- getLine
   print "Who is the Author?"
   author <- getLine
   print (createEmail recipient title author)
```

You should be all set to compile, but it's always a good idea to test in GHCi first:

```
GHCi> main
"Who is the email for?"
  Happy Reader
"What is the Title?"
```

```
  Learn Haskell
"Who is the Author?"
  Will Kurt
"Dear Happy Reader,\nThanks for buying Learn Haskell.\nThanks,\nWill Kurt"
```

It looks like all your pieces are working together, and you were able to play with them each individually to make sure they worked as expected. Finally, you can compile your program:

```
$ ghc first_prog.hs
[1 of 1] Compiling Main             ( first_prog.hs, first_prog.o )
Linking first_prog ...
$ ./first_prog
"Who is the email for?"
Happy Reader
"What is the Title?"
Learn Haskell
"Who is the Author?"
Will Kurt
"Dear Happy Reader,\nThanks for buying Learn Haskell.\nThanks,\nWill Kurt"
```

You've just finished your first successful Haskell program. With your basic workflow understood, you can now dive into the amazing world of Haskell!

Summary

In this lesson, our objective was to get you started with Haskell. You started by installing the Haskell Platform, which bundles together the tools you'll be using through this book. These tools include GHC, Haskell's compiler; GHCi, the interactive interpreter for Haskell; and stack, a build tool you'll use later in the book. The rest of this lesson covered the basics of writing, refactoring, interacting with, and compiling Haskell programs. Let's see if you got this.

Q1.1 In GHCi, find out what 2^{123} is.

Q1.2 Modify the text in each of the functions in first_prog.hs, test them out in GHCi while you do this, and, finally, compile a new version of your email templating program so that the executable is named *email*.

13 LESSON

Lesson 13 from *Get Programming with Haskell* by Will Kurt

TYPE CLASSES

After reading lesson 13, you'll be able to

- Understand the basics of type classes
- Read type class definitions
- Use common type classes: `Num`, `Show`, `Eq`, `Ord`, and `Bounded`

In this lesson, you're going to look an important abstraction in Haskell's type system: type classes. Type classes allow you to group types based on shared behavior. At first glance, type classes are similar to interfaces in most object-oriented programming languages. A type class states which functions a type must support in the same way that an interface specifies which methods a class must support. But type classes play a much more important role in Haskell than interfaces do in languages such as Java and C#. The major difference is that as you dive deeper into Haskell, you'll see that type classes typically require you to think in increasingly more powerful forms of abstraction. In many ways, type classes are the heart of Haskell programming.

Consider this You've written the function `inc` to increment a value a few times as a sample function. But how can you write an incrementing function that works with the wide range of possible numbers you've seen? Frustratingly enough, in unit 1, without specifying types, you could do this. How can you write the type signature of an `inc` function that works on all numbers?

13.1 Further exploring types

At this point, you've seen quite a few type signatures and even built some nontrivial types of your own. One of the best ways to learn about various Haskell types is to use the `:t` (or more verbose `:type`) command in GHCi to inspect the type of function you find in the wild. When you first wrote `simple`, you did so without a type signature:

```
simple x = x
```

If you wanted to know what type this function was, you could load it into GHCi and use `:t`:

```
GHCi> :t simple
simple :: t -> t
```

You could do the same thing for the lambda version of `simple`:

```
GHCi> :t (\x -> x)
(\x -> x) :: r -> r
```

Quick check 13.1 Find the type of the following:

```
aList = ["cat","dog","mouse"]
```

If you start exploring types this way, you'll almost immediately come across some things you haven't seen yet. Take, for example, something as simple as addition:

```
GHCi> :t (+)
(+) :: Num a => a -> a -> a
```

With all the time you've spent so far looking at types, something as simple as addition trips you up! The big mystery is the `Num a =>` part.

13.2 Type classes

What you've encountered here is your first type class! *Type classes* in Haskell are a way of describing groups of types that all behave in the same way. If you're familiar with Java or C#, type classes may remind you of interfaces. When you see `Num a`, the best way

QC 13.1 answer

```
aList = ["cat","dog","mouse"]
GHCi> :t aList
aList :: [[Char]]
```

to understand that statement is to say that there's some type `a` of class `Num`. But what does it mean to be part of type class `Num`? `Num` is a type class generalizing the idea of a number. All things of class `Num` must have a function (`+`) defined on them. There are other functions in the type class as well. One of the most valuable GHCi tools is `:info`, which provides information about types and type classes. If you use `:info` on `Num`, you get the following (partial) output.

Listing 13.1 `Num` type class definition

```
GHCi> :info Num
class Num a where
   (+) :: a -> a -> a
   (-) :: a -> a -> a
   (*) :: a -> a -> a
   negate :: a -> a
   abs :: a -> a
   signum :: a -> a
```

What `:info` is showing is the definition of the type class. The definition is a list of functions that all members of the class must implement, along with the type signatures of those functions. The family of functions that describe a number is `+`, `-`, `*`, `negate`, `abs`, and `signum` (gives the sign of a number). Each type signature shows the same type variable `a` for all arguments and the output. None of these functions can return a different type than it takes as an argument. For example, you can't add two `Int`s and get a `Double`.

Quick check 13.2 Why isn't division included in the list of functions needed for a `Num`?

13.3 The benefits of type classes

Why do you need type classes at all? So far in Haskell, each function you've defined works for only one specific set of types. Without type classes, you'd need a different name for each function that adds a different type of value. You do have type variables, but they're too flexible. For example, say you define `myAdd` with the following type signature:

```
myAdd :: a -> a -> a
```

QC 13.2 answer Because division with (`/`) isn't defined on all cases of `Num`.

Then you'd need the ability to manually check that you were adding only the types it makes sense to add (which isn't possible in Haskell).

Type classes also allow you to define functions on a variety of types that you can't even think of. Suppose you want to write an `addThenDouble` function like the following.

Listing 13.2 Using type classes: `addThenDouble`

```
addThenDouble :: Num a => a -> a -> a
addThenDouble x y = (x + y)*2
```

Because you use the `Num` type class, this code will automatically work not only on `Int` and `Double`, but also on anything that another programmer has written and implemented the `Num` type class for. If you end up interacting with a Roman Numerals library, as long as the author has implemented the `Num` type class, this function will still work!

13.4 Defining a type class

The output you got from GHCi for `Num` is the literal definition of the type class. Type class definitions have the structure illustrated in figure 13.1.

In the definition of `Num`, you see plenty of type variables. Nearly all functions required in any type class definition will be expressed in terms of type variables, because by definition you're describing an entire class of types. When you define a type class, you're doing so precisely because you don't want your functions to be tied to a single type. One way of thinking of type classes is as a constraint on the categories of types that a type variable can represent.

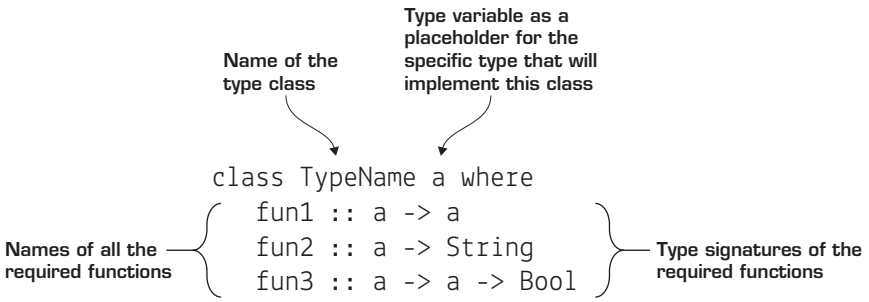


Figure 13.1 Structure of a type class definition

To help solidify the idea, you'll write a simple type class of your own. Because you're learning Haskell, a great type class to have is `Describable`. Any type that's an instance of your `Describable` type class can describe itself to you in plain English. So you require only one function, which is `describe`. For whatever type you have, if it's `Describable`, calling `describe` on an instance of the type will tell you all about it. For example, if `Bool` were `Describable`, you'd expect this:

```
GHCi> describe True
"A member of the Bool class, True is opposite of False"
GHCi> describe False
"A member of the Bool class, False is the opposite of True"
```

And if you wanted to describe an `Int`, you might expect this:

```
GHCi> describe (6 :: Int) "A member of the Int class, the number after 5 and
          before 7"
```

At this point, you won't worry about implementing a type class (you'll do that in the next lesson)—only defining it. You know that you require only one function, which is `describe`. The only other thing you need to worry about is the type signature of that function. In each case, the argument for the function is whatever type has implemented your type class, and the result is always a string. So you need to use a type variable for the first type and a string for the return value. You can put this all together and define your type class as follows.

Listing 13.3 Defining your own type class: `Describable`

```
class Describable a where
   describe :: a -> String
```

And that's it! If you wanted to, you could build a much larger group of tools that use this type class to provide automatic documentation for your code, or generate tutorials for you.

13.5 Common type classes

Haskell defines many type classes for your convenience, which you'll learn about in the course of this book. In this section, you'll look at four more of the most basic: `Ord`, `Eq`, `Bounded`, and `Show`.

13.6 The Ord and Eq type classes

Let's look at another easy operator, *greater than* (>):

```
GHCi> :t (>)
(>) :: Ord a => a -> a -> Bool
```

Here's a new type class, `Ord`! This type signature says, "Take any two of the same types that implement `Ord`, and return a Boolean." `Ord` represents all of the things in the universe that can be compared and ordered. Numbers can be compared, but so can strings and many other things. Here's the list of functions that `Ord` defines.

Listing 13.4 `Ord` type class requires `Eq` type class

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Of course, Haskell has to make things complicated. Notice that right in the class definition there's another type class! In this case, it's the type class `Eq`. Before you can understand `Ord`, you should look at `Eq`.

Listing 13.5 `Eq` type class generalizes the idea of equality

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The `Eq` type class needs only two functions: == and /=. If you can tell that two types are equal or not equal, that type belongs in the `Eq` type class. This explains why the `Ord` type class includes the `Eq` type class in its definition. To say that something is ordered, clearly you need to be able to say that things of that type can be equal. But the inverse isn't true. We can describe many things by saying, "These two things are equal," but not "This is better than that one." You may love vanilla ice cream more than chocolate, and I might

love chocolate more than vanilla. You and I can agree that two vanilla ice-cream cones are the same, but we can't agree on the order of a chocolate and vanilla cone. So if you created an `IceCream` type, you could implement `Eq`, but not `Ord`.

13.6.1 Bounded

In lesson 11, we mentioned the difference between the `Int` and `Integer` types. It turns out this difference is also captured by a type class. The `:info` command was useful for learning about type classes, but it's also helpful in learning about types. If you use `:info` on `Int`, you get a list of all the type classes that `Int` is a member of:

```
GHCi> :info Int
data Int = GHC.Types.I# GHC.Prim.Int#    -- Defined in 'GHC.Types'
instance Bounded Int -- Defined in 'GHC.Enum'
instance Enum Int -- Defined in 'GHC.Enum'
instance Eq Int -- Defined in 'GHC.Classes'
instance Integral Int -- Defined in 'GHC.Real'
instance Num Int -- Defined in 'GHC.Num'
instance Ord Int -- Defined in 'GHC.Classes'
instance Read Int -- Defined in 'GHC.Read'
instance Real Int -- Defined in 'GHC.Real'
instance Show Int -- Defined in 'GHC.Show'
```

You can do the same thing for the `Integer` type. If you did, you'd find there's a single difference between the two types. `Int` is an instance of the `Bounded` type class, and `Integer` isn't. Understanding the type classes involved in a type can go a long way toward helping you understand how a type behaves. `Bounded` is another simple type class (most are), which requires only two functions. Here's the definition of `Bounded`.

Listing 13.6 `Bounded` type class requires values but no functions

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

Members of `Bounded` must provide a way to get their `upper` and `lower` bounds. What's interesting is that `minBound` and `maxBounds` aren't functions but values! They take no arguments but are just a value of whatever type they happen to be. Both `Char` and `Int` are members of the `Bounded` type class, so you never have to guess the upper and lower bounds for using these values:

```
GHCi> minBound :: Int
-9223372036854775808
GHCi> maxBound :: Int
9223372036854775807
GHCi> minBound :: Char
'\NUL'
GHCi> maxBound :: Char
'\1114111'
```

13.6.2 Show

We mentioned the functions `show` and `read` in lesson 11. `Show` and `Read` are incredibly useful type classes that make the `show` and `read` functions possible. Aside from two special cases for specific types, `Show` implements just one important function: `show`.

Listing 13.7 `Show` type class definition

```
class Show a where
  show :: a -> String
```

The `show` function turns a value into a `String`. Any type that implements `Show` can be printed. You've been making much heavier use of `show` than you might have realized. Every time a value is printed in GHCi, it's printed because it's a member of the `Show` type class. As a counter example, let's define your `Icecream` type but not implement `show`.

Listing 13.8 Defining the `Icecream` type

```
data Icecream = Chocolate | Vanilla
```

`Icecream` is nearly identical to `Bool`, but `Bool` implements `Show`. Look what happens when you type the constructors for these into GHCi:

```
GHCi> True
True
GHCi> False
False
GHCi> Chocolate
<interactive>:404:1:
No instance for (Show Icecream) arising from a use of 'print'
In a stmt of an interactive GHCi command: print it
```

You get an error because Haskell has no idea how to turn your data constructors into strings. Every value that you've seen printed in GHCi has happened because of the `Show` type class.

13.7 Deriving type classes

For your `Icecream` type class, it's a bit annoying that you have to implement `Show`. After all, `Icecream` is just like `Bool`, so why can't you have Haskell be smart about it and do what it does with `Bool`? In `Bool`, all that happens is that the data constructors are printed out. It just so happens that Haskell is rather smart! When you define a type, Haskell can do its best to automatically derive a type class. Here's the syntax for defining your `Icecream` type but deriving `Show`.

Listing 13.9 The `Icecream` type deriving the `Show` type class

```
data Icecream = Chocolate | Vanilla deriving (Show)
```

Now you can go back to GHCi, and everything works great:

```
GHCi> Chocolate
Chocolate
GHCi> Vanilla
Vanilla
```

Many of the more popular type classes have a reasonable default implementation. You can also add the `Eq` type class:

```
data Icecream = Chocolate | Vanilla deriving (Show, Eq, Ord)
```

And again you can use `GHCi` to show that you can see whether two flavors of `Icecream` are identical:

```
GHCi> Vanilla == Vanilla
True
GHCi> Chocolate == Vanilla
False
GHCi> Chocolate /= Vanilla
True
```

In the next lesson, you'll look more closely at how to implement your own type classes, as Haskell isn't always able to guess your true intentions.

Quick check 13.3 See which flavor Haskell thinks is superior by deriving the `Ord` type class.

Summary

In this lesson, our objective was to teach you the basics of type classes. All of the type classes we covered should seem familiar to users of object-oriented languages such as Java and C# that support interfaces. The type classes you saw make it easy to apply one function to a wide variety of types. This makes testing for equality, sorting data, and converting data to a string much easier. Additionally, you saw that Haskell is able to automatically implement type classes for you in some cases by using the `deriving` keyword. Let's see if you got this.

Q13.1 If you ran the `:info` examples, you likely noticed that the type `Word` has come up a few times. Without looking at external resources, use `:info` to explore `Word` and the relevant type classes to come up with your own explanation for the `Word` type. How is it different from `Int`?

Q13.2 One type class we didn't discuss is `Enum`. Use `:info` to look at the definition of this type class, as well as example members. Now consider `Int`, which is an instance of both `Enum` and `Bounded`. Given the following definition of `inc`:

```
inc :: Int -> Int
inc x = x + 1
```

and the `succ` function required by `Enum`, what's the difference between `inc` and `succ` for `Int`?

Q13.3 Write the following function that works just like `succ` on `Bounded` types but can be called an unlimited number of times without error. The function will work like `inc` in the preceding example but works on a wider range of types, including types that aren't members of `Num`:

```
cycleSucc :: (Bounded a, Enum a, ? a) => a -> a
cycleSucc n = ?
```

Your definition will include functions/values from `Bounded`, `Enum`, and the mystery type class. Make a note of where each of these three (or more) functions/values comes from.

QC 13.3 answer If you add `deriving Ord` to your definition of `Icecream`, Haskell defaults to the order of the data constructors for determining `Ord`. So `Vanilla` will be greater than `Chocolate`.

14

LESSON

Lesson 14 from *Get Programming with Haskell* by Will Kurt

USING TYPE CLASSES

After reading lesson 14, you'll be able to

- Implement your own type classes
- Understand polymorphism in Haskell
- Know when to use `deriving`
- Search for documentation with Hackage and Hoogle

In lesson 13, you got your first look at type classes, which are Haskell's way of grouping types by common behaviors they share. In this lesson, you'll take a deeper look at how to implement existing type classes. This will allow you to write new types that take advantage of a wide range of existing functions.

Consider this You have a data type consisting of data constructors for New England states:

```
data NewEngland = ME | VT | NH | MA | RI | CT
```

You want to be able to display them by their full name by using `Show`. You can easily display their abbreviations by deriving `show`, but there's no obvious way to create your own version of `show`. How can you make your `NewEngland` type display the full state name by using `show`?

14.1 A type in need of classes

You'll start by modeling a six-sided die. A good default implementation is a type similar to `Bool`, only with six values instead of two. You'll name your data constructors `S1` through `S6` to represent each of the six sides.

Listing 14.1 Defining the `SixSidedDie` data type

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6
```

Next you want to implement some useful type classes. Perhaps the most important type class to implement is `Show`, because you'll nearly always want to have an easy way to display instances of your type, especially in GHCi. In lesson 13, we mentioned that you could add the `deriving` keyword to automatically create instances of a class. You could define `SixSidedDie` this way and call it a day.

Listing 14.2 The `SixSidedDie` type deriving `Show`

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6 deriving (Show)
```

If you were to use this type in GHCi, you'd get a simple text version of your data constructors back when you type them:

```
GHCi> S1
S1
GHCi> S2
S2
GHCi> S3
S3
GHCi> S4
S4
```

This is a bit boring because you're just printing out your data constructors, which are more meaningful from an implementation standpoint than they are readable. Instead, let's print out the English word for each number.

14.2 Implementing Show

To do this, you have to implement your first type class, `Show`. There's only one function (or in the case of type classes, we call these *methods*) that you have to implement, `show`. Here's how to implement your type class.

Listing 14.3 Creating an instance of `Show` for `SixSidedDie`

```
instance Show SixSidedDie where
   show S1 = "one"
   show S2 = "two"
   show S3 = "three"
   show S4 = "four"
   show S5 = "five"
   show S6 = "six"
```

And that's it! Now you can return to GHCi and much more interesting output than you would with `deriving`:

```
GHCi> S1
one
GHCi> S2
two
GHCi> S6
six
```

Quick check 14.1 Rewrite this definition of `show` to print the numerals 1–6 instead.

QC 14.1 answer

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6

instance Show SixSidedDie where
   show S1 = "I"
   show S2 = "II"
   show S3 = "III"
   show S4 = "IV"
   show S5 = "V"
   show S6 = "VI"
```

14.3 Type classes and polymorphism

One question that might come up is, why do you have to define `show` this way? Why do you need to declare an instance of a type class? Surprisingly, if you remove your early instance declaration, the following code will compile just fine.

Listing 14.4 Incorrect attempt to implement `show` for `SixSidedDie`

```
show :: SixSidedDie -> String
show S1 = "one"
show S2 = "two"
show S3 = "three"
show S4 = "four"
show S5 = "five"
show S6 = "six"
```

But if you load this code into GHCi, you get two problems. First, GHCi no longer can print your data constructors by default. Second, even if you manually use `show`, you get an error:

```
"Ambiguous occurrence 'show'"
```

You haven't learned about Haskell's module system yet, but the issue Haskell has is that the definition you just wrote for `show` is conflicting with another that's defined by the type class. You can see the real problem when you create a `TwoSidedDie` type and attempt to write `show` for it.

Listing 14.5 Demonstrating the need for polymorphism defining `show` for `TwoSidedDie`

```
data TwoSidedDie = One | Two

show :: TwoSidedDie -> String
show One = "one"
show Two = "two"
```

The error you get now is as follows:

```
Multiple declarations of 'show'
```

The problem is that by default you'd like to have more than one behavior for `show`, depending on the type you're using. What you're looking for here is called *polymorphism*. Polymorphism means that the same function behaves differently depending on

the type of data it's working with. Polymorphism is important in object-oriented programming and equally so in Haskell. The OOP equivalent to `show` would be a `toString` method, one that's common among any classes that can be turned into a string. Type classes are the way you use polymorphism in Haskell, as shown in figure 14.1.

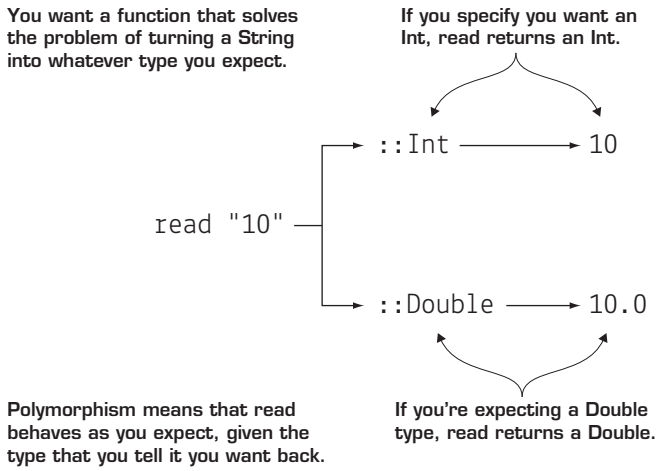


Figure 14.1 Visualizing polymorphism for `read`

14.4 Default implementation and minimum complete definitions

Now that you can produce fun strings for your `SixSidedDie`, it'd be useful to determine that two dice are the same. This means that you have to implement the `Eq` class. This is also useful because `Eq` is the *superclass* of `Ord`. You touched on this relationship briefly in lesson 13 without giving it a name. To say that `Eq` is a superclass of `Ord` means that every instance of `Ord` must also be an instance of `Eq`. Ultimately, you'd like to compare `SixSidedDie` data constructors, which means implementing `Ord`, so first you need to implement `Eq`. Using the `:info` command in GHCi, you can bring up the class definition for `Eq`:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

You have to implement only two methods: the Equals method `(==)` and the Not Equals method `(/=)`. Given how smart Haskell has been so far, this should seem like more work than makes sense. After all, if you know the definition of `(==)`, the definition of `(/=)` is

not (==). Sure, there may be some exceptions to this, but it seems that in the vast majority of cases, if you know either one, then you can determine the other.

It turns out that Haskell is smart enough to figure this out. Type classes can have *default implementations* of methods. If you define (==), Haskell can figure out what (/=) means without any help.

Listing 14.6 Implementing an instance of `Eq` for `SixSidedDie`

```
instance Eq SixSidedDie where
   (==) S6 S6 = True
   (==) S5 S5 = True
   (==) S4 S4 = True
   (==) S3 S3 = True
   (==) S2 S2 = True
   (==) S1 S1 = True
   (==) _ _ = False
```

In GHCi, you'll see that (/=) works automatically!

```
GHCi> S6 == S6
True
GHCi> S6 == S5
False
GHCi> S5 == S6
False
GHCi> S5 /= S6
True
GHCi> S6 /= S6
False
```

This is useful, but how in the world are you supposed to know which methods you need to implement? The `:info` command is a great source of information right at your fingertips, but it isn't complete documentation. A source of more thorough information is *Hackage*, Haskell's centralized package library. Hackage can be found on the web at https://hackage.haskell.org. If you go to `Eq`'s page on Hackage (https://hackage.haskell.org/package/base/docs/Data-Eq.html), you get much more info on `Eq` (probably more than you could ever want!). For our purposes, the most important part is a section called "Minimum complete definition." For `Eq`, you find the following:

```
(==) | (/=)
```

This is much more helpful! To implement the `Eq` type class, all you have to define is either `(==)` or `(/=)`. Just as in data declarations, `|` means *or*. If you provide either one of these options, Haskell can work out the rest for you.

Hackage and Hoogle

Although Hackage may be the central repository for Haskell information, you might find it a pain to search for specific types. To solve this, Hackage can be searched via a truly amazing interface called *Hoogle*. Hoogle can be found at https://www.haskell.org/hoogle. Hoogle allows you to search by types and type signatures. For example, if you search `a -> String`, you'll get results for `show` along with a variety of other functions. Hoogle alone is enough to make you love Haskell's type system.

Quick check 14.2 Use Hoogle to search for the `RealFrac` type class. What's its minimal complete definition?

14.5 Implementing Ord

One of the most important features of dice is that there's an order to their sides. `Ord` defines a handful of useful functions for comparing a type:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Luckily, on Hackage you can find that only the `compare` method needs to be implemented. The `compare` method takes two values of your type and returns `Ordering`. This is a type you

QC 14.2 answer Go to http://hackage.haskell.org/package/base/docs/Prelude.html#t:RealFrac. The minimal complete definition is `properFraction`.

saw briefly when you learned about `sort` in lesson 4. `Ordering` is just like `Bool`, except it has three data constructors. Here's its definition:

```
data Ordering = LT | EQ | GT
```

The following is a partial definition of `compare`.

Listing 14.7 Partial definition of `compare` for `SixSidedDie`

```
instance Ord SixSidedDie where
   compare S6 S6 = EQ
   compare S6 _ = GT
   compare _ S6 = LT
   compare S5 S5 = EQ
   compare S5 _ = GT
   compare _ S5 = LT
```

Even with clever uses of pattern matching, filling out this complete definition would be a lot of work. Imagine how large this definition would be for a 60-sided die!

Quick check 14.3 Write out the patterns for the case of `S4`.

14.6 To derive or not to derive?

So far, every class you've seen has been *derivable*, meaning that you can use the deriving keyword to automatically implement these for your new type definition. It's common for programming languages to offer default implementations for things such as an `.equals` method (which is often too minimal to be useful). The question is, how much should you rely on Haskell to derive your type classes versus doing it yourself?

QC 14.3 answer

```
compare S4 S4 = EQ
compare _ S4 = LT
```

Note: Because of pattern matching, the case of `compare S5 S4` and `compare S6 S4` will already be matched.

```
compare S4 _ = GT
```

Let's look at `Ord`. In this case, it's wiser to use `deriving (Ord)`, which works much better in cases of simple types. The default behavior when deriving `Ord` is to use the order that the data constructors are defined. For example, consider the following listing.

Listing 14.8 How deriving `Ord` is determined

```
data Test1 = AA | ZZ deriving (Eq, Ord)
data Test2 = ZZZ | AAA deriving (Eq, Ord)
```

In GHCi, you can see the following:

```
GHCi> AA < ZZ
True
GHCi> AA > ZZ
False
GHCi> AAA > ZZZ
True
GHCi> AAA < ZZZ
False
```

Quick check 14.4 Rewrite `SixSidedDie` to derive both `Eq` and `Ord`.

With `Ord`, using the `deriving` keyword saves you from writing a lot of unnecessary and potentially buggy code.

An even stronger case for using `deriving` when you can is `Enum`. The `Enum` type allows you to represent your dice sides as an enumerated list of constants. This is essentially what we think of when we think of a die, so it'll be useful. Here's the definition:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
```

QC 14.4 answer

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6 deriving (Show,Eq,Ord)
```

```
enumFromThen :: a -> a -> [a]
enumFromTo :: a -> a -> [a]
enumFromThenTo :: a -> a -> a -> [a]
```

Once again, you're saved by having to implement only two methods: `toEnum` and `fromEnum`. These methods translate your `Enum` values to and from an `Int`. Here's the implementation.

Listing 14.9 Implementing `Enum` for `SixSidedDie` (errors with implementation)

```
instance Enum SixSidedDie where
   toEnum 0 = S1
   toEnum 1 = S2
   toEnum 2 = S3
   toEnum 3 = S4
   toEnum 4 = S5
   toEnum 5 = S6
   toEnum _ = error "No such value"

   fromEnum S1 = 0
   fromEnum S2 = 1
   fromEnum S3 = 2
   fromEnum S4 = 3
   fromEnum S5 = 4
   fromEnum S6 = 5
```

Now you can see some of the practical benefits of `Enum`. For starters, you can now generate lists of your `SixSidedDie` just as you can other values such as `Int` and `Char`:

```
GHCi> [S1 .. S6]
[one,two,three,four,five,six]
GHCi> [S2,S4 .. S6]
[two,four,six]
GHCi> [S4 .. S6]
[four,five,six]
```

This is great so far, but what happens when you create a list with no end?

```
GHCi> [S1 .. ]
[one,two,three,four,five,six,*** Exception: No such value
```

Yikes! You get an error because you didn't handle the case of having a missing value.

But if you had just derived your type class, this wouldn't be a problem:

```
data SixSidedDie = S1 | S2 | S3 | S4 | S5 | S6 deriving (Enum)
GHCi> [S1 .. ]
[one,two,three,four,five,six]
```

Haskell is pretty magical when it comes to deriving type classes. In general, if you don't have a good reason to implement your own, deriving is not only easier, but also often better.

14.7 Type classes for more-complex types

In lesson 4, we demonstrated that you can use first-class functions to properly order something like a tuple of names.

Listing 14.10 Using a type synonym for `Name`

```
type Name = (String,String)

names :: [Name]
names = [ ("Emil","Cioran")
        , ("Eugene","Thacker")
        , ("Friedrich","Nietzsche")]
```

As you may remember, you have a problem when these are sorted:

```
GHCi> import Data.List
GHCi> sort names
[("Emil","Cioran"),("Eugene","Thacker"),("Friedrich","Nietzsche")]
```

The good thing is that clearly your tuples automatically derive `Ord`, because they're sorted well. Unfortunately, they aren't sorted the way you'd like them to be, by last name and then first name. In lesson 4, you used a first-class function and passed it to `sortBy`, but that's annoying to do more than once. Clearly, you can implement your own custom `Ord` for `Name`.

Listing 14.11 Attempt to implement `Ord` for a type synonym

```
instance Ord Name where
   compare (f1,l1) (f2,l2) = compare (l1,f1) (l2,f2)
```

But when you try to load this code, you get an error! This is because to Haskell, `Name` is identical to `(String, String)`, and, as you've seen, Haskell already knows how to sort these. To solve these issues, you need create a new data type. You can do this by using the data as before.

Listing 14.12 Defining a new type Name using data

```
data Name = Name (String, String) deriving (Show, Eq)
```

Here the need for data constructors becomes clear. For Haskell, they're a way to note, "This tuple is special from the others." Now that you have this, you can implement your custom `Ord`.

Listing 14.13 Correct implementation of Ord for Name type

```
instance Ord Name where
   compare (Name (f1,l1)) (Name (f2,l2)) = compare (l1,f1) (l2,f2)
```

Notice that you're able to exploit the fact that Haskell derives `Ord` on the `(String, String)` tuple to make implementing your custom `compare` much easier:

```
names :: [Name]
names = [Name ("Emil","Cioran")
         , Name ("Eugene","Thacker")
         , Name ("Friedrich","Nietzsche")]
```

Now your names are sorted as expected:

```
GHCi> import Data.List
GHCi> sort names
[Name ("Emil","Cioran"),Name ("Friedrich","Nietzsche"),
➥Name ("Eugene","Thacker")]
```

Creating types with newtype

When looking at our type definition for `Name`, you find an interesting case in which you'd like to use a type synonym, but need to define a data type in order to make your type an instance of a type class. Haskell has a preferred method of doing this: using the `newtype` keyword. Here's an example of the definition of `Name` using `newtype`:

```
newtype Name = Name (String, String) deriving (Show, Eq)
```

In cases like this, `newtype` is often more efficient than using `data`. Any type that you can define with `newtype`, you can also define using `data`. But the opposite isn't true. Types defined with `newtype` can have only one type constructor and one type (in the case of `Name`, it's `Tuple`). In most cases, when you need a type constructor to make a type synonym more powerful, `newtype` is going to be the preferred method.

For simplicity, we'll stick to creating types with `data` throughout this book.

14.8 Type class roadmap

Figure 14.2 shows the type classes that are defined in Haskell's standard library. Arrows from one class to another indicate a superclass relationship. This unit has covered most of the basic type classes. In unit 3, you'll start exploring the more abstract type classes `Semigroup` and `Monoid`, and you'll start to see how different type classes can be from interfaces. In unit 5, you'll look at a family of type classes—`Functor`, `Applicative`, and `Monad`—that provide a way to model the context of a computation. Although this last group is particularly challenging to learn, it also allows for some of Haskell's most powerful abstractions.

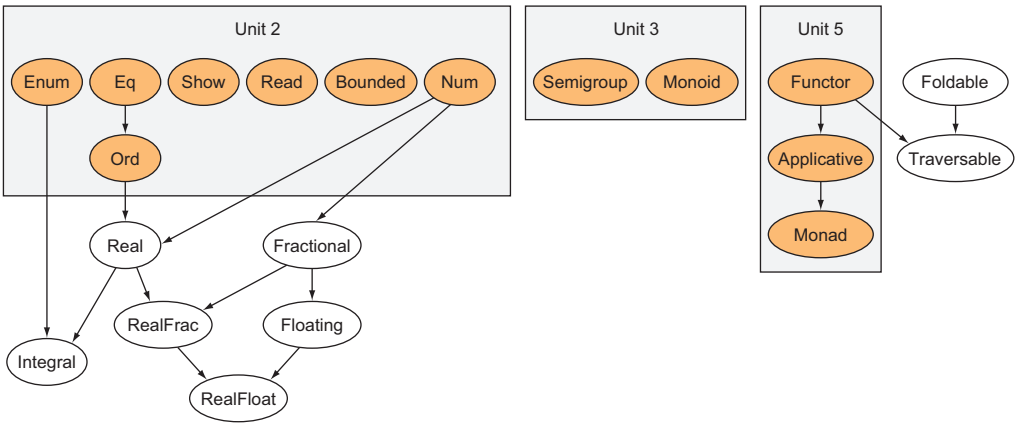


Figure 14.2 Type class road map

Summary

In this lesson, our objective was to dive deeper into Haskell's type classes. You learned how to read type class definitions as well as how to make types an instance of a type class beyond simply using the `deriving` keyword. You also learned when it's best to use `deriving` and when you should write your own instances of a type class. Let's see if you got this.

Q14.1 Note that `Enum` doesn't require either `Ord` or `Eq`, even though it maps types to `Int` values (which implement both `Ord` and `Eq`). Ignoring the fact that you can easily use `deriving` for `Eq` and `Ord`, use the derived implementation of `Enum` to make manually defining `Eq` and `Ord` much easier.

Q14.2 Define a five-sided die (`FiveSidedDie` type). Then define a type class named `Die` and at least one method that would be useful to have for a die. Also include superclasses you think make sense for a die. Finally, make your `FiveSidedDie` an instance of `Die`.

LESSON 17

Lesson 17 from *Get Programming with Haskell* by Will Kurt

DESIGN BY COMPOSITION—SEMIGROUPS AND MONOIDS

After reading lesson 17, you'll be able to

- Create new functions with function composition
- Use `Semigroup` to mix colors
- Learn how to use guards in code
- Solve probability problems with `Monoid`

In the preceding lesson, you looked at how sum types allow you to think outside the typical hierarchical design patterns present in most programming languages. Another important way that Haskell diverges from traditional software design is with the idea of composability. *Composability* means that you create something new by combining two like things.

What does it mean to *combine* two things? Here are some examples: you can concatenate two lists and get a new list, you can combine two documents and get a new document, and you can mix two colors and get a new color. In many programming languages, each of these methods of combining types would have its own unique operator or function. In much the same way that nearly every programming language offers a standard way to convert a type to a string, Haskell offers a standard way to combine instances of the same type together.

Consider this So far, when you've combined multiple strings, you've used ++. This can get tedious for larger strings:

```
"this" ++ " " ++ "is" ++ " " ++ "a" ++ " " ++ "bit" ++ " " ++ "much"
```

Is there a better way to solve this?

17.1 Intro to composability—combining functions

Before diving into combining types, let's look at something more fundamental: combining functions. A special higher-order function that's just a period (called *compose*) takes two functions as arguments. Using function composition is particularly helpful for combining functions on the fly in a readable way. Here are some examples of functions that can easily be expressed using function composition.

Listing 17.1 Examples of using function composition to create functions

```
myLast :: [a] -> a
myLast = head . reverse

myMin :: Ord a => [a] -> a
myMin = head . sort

myMax :: Ord a => [a] -> a
myMax = myLast . sort

myAll :: (a -> Bool) -> [a] -> Bool
myAll testFunc = (foldr (&&) True) . (map testFunc)
```

Using sort requires the Data.List module to be imported.

myAll tests that a property is true of all items in a list.

Quick check 17.1 Implement `myAny` by using function composition. `myAny` tests that a property is `True` for at least one value in the list.

QC 17.1 answer

```
myAny :: (a -> Bool) -> [a] -> Bool
myAny testFunc = (foldr (||) False) . (map testFunc)
```

Here's an example:

```
GHCi> myAny even [1,2,3]
True
```

In many cases where you'd use a lambda expression to create a quick function, function composition will be more efficient and easier to read.

17.2 Combining like types: Semigroups

To explore composability further, let's look at a remarkably simple type class called `Semigroup`. To do this, you need to import `Data.Semigroup` at the top of your file (Lesson17.hs for this lesson).

The `Semigroup` class has only one important method you need, the `<>` operator. You can think of `<>` as an operator for combining instances of the same type. You can trivially implement `Semigroup` for `Integer` by defining `<>` as `+`.

Listing 17.2 `Semigroup` for `Integer`

```
instance Semigroup Integer where
  (<>) x y = x + y
```

You use the "instance" keyword to make Integer an instance of the Semigroup type class.

You define the <> operator as simple addition.

This may seem all too trivial, but it's important to think about what this means. Here's the type signature for `(<>)`:

```
(<>) :: Semigroup a => a -> a -> a
```

This simple signature is the heart of the idea of composability; you can take two like things and combine them to get a new thing of the same type.

Quick check 17.2 Can you use `(/)` to make `Int` a `Semigroup`?

17.2.1 The Color Semigroup

Initially, it might seem like this concept would be useful only for mathematics, but we're all familiar with this idea from an early age. The most well-known example of this is

QC 17.2 answer No, because division doesn't always return an Int type, which violates the rule.

adding colors. As most children experience, we can combine basic colors to get a new color. For example:

- Blue and yellow make green.
- Red and yellow make orange.
- Blue and red make purple.

You can easily use types to represent this problem of mixing colors. First, you need a simple sum type of the colors.

Listing 17.3 Defining the `Color` type

```
data Color = Red |
   Yellow |
   Blue |
   Green |
   Purple |
   Orange |
   Brown deriving (Show,Eq)
```

Next you can implement `Semigroup` for your `Color` type.

Listing 17.4 Implementing `Semigroup` for `Color` v1

```
instance Semigroup Color where
   (<>) Red Blue = Purple
   (<>) Blue Red = Purple
   (<>) Yellow Blue = Green
   (<>) Blue Yellow = Green
   (<>) Yellow Red = Orange
   (<>) Red Yellow = Orange
   (<>) a b = if a == b
              then a
              else Brown
```

Now you can play with colors just as you did when you smeared your fingers in paint as a kid!

```
GHCi> Red <> Yellow
Orange
GHCi> Red <> Blue
```

```
Purple
GHCi> Green <> Purple
Brown
```

This works great, but you get an interesting problem when you add more than two colors. You want your color mixing to be associative. *Associative* means that the order in which you apply your <> operator doesn't matter. For numbers, this means that 1 + (2 + 3) = (1 + 2) + 3. As you can see, your colors clearly aren't associative:

```
GHCi> (Green <> Blue) <> Yellow
Brown
GHCi> Green <> (Blue <> Yellow)
Green
```

Not only does this rule about associativity make intuitive sense (mixing colors in any order should give you the same color), but this is formally required of the `Semigroup` type class. This can be one of the more confusing parts of the more advanced type classes we cover in this unit. Many of them have *type class laws* that require certain behavior. Unfortunately, the Haskell compiler can't enforce these. The best advice is to always carefully read the Hackage documentation (https://hackage.haskell.org/) whenever you implement a nontrivial type class on your own.

17.2.2 Making Color associative and using guards

You can fix this issue by making it so that if one color is used to make another, combining them yields the composite color. So purple plus red is still purple. You could approach this problem by listing out a large number of pattern-matching rules comparing each possibility. But this solution would be long. Instead, you'll use the Haskell feature called guards. *Guards* work much like pattern matching, but they allow you to do some computation on the arguments you're going to compare. Figure 17.1 shows an example of a function using guards.

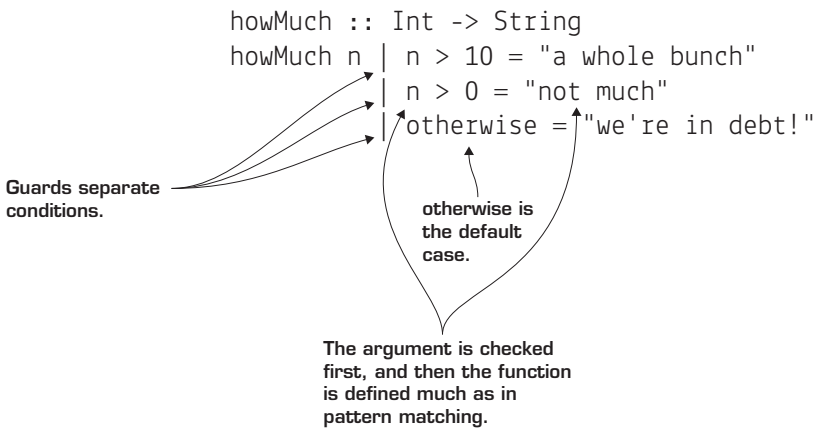


Figure 17.1 Using guards in `howMuch`

With an understanding of guards, you can rewrite your instance of `Semigroup` for `Color` so that you adhere to the type class laws for semigroups.

Listing 17.5 Reimplementing `Semigroup` for `Color` to support associativity

```
instance Semigroup Color where
   (<>) Red Blue = Purple
   (<>) Blue Red = Purple
   (<>) Yellow Blue = Green
   (<>) Blue Yellow = Green
   (<>) Yellow Red = Orange
   (<>) Red Yellow = Orange
   (<>) a b | a == b = a
            | all (`elem` [Red,Blue,Purple]) [a,b] = Purple
            | all (`elem` [Blue,Yellow,Green]) [a,b] = Green
            | all (`elem` [Red,Yellow,Orange]) [a,b] = Orange
            | otherwise = Brown
```

As you can see, now the problem is fixed:

```
GHCi> (Green <> Blue) <> Yellow
Green
GHCi> Green <> (Blue <> Yellow)
Green
```

Type class laws are important because any other code that uses an instance of a type class will assume that they're upheld.

Quick check 17.3 Does your implementation of `Semigroup` for `Integers` support associativity?

In the real world, there are many ways to make a new thing from two things of the same type. Imagine the following possibilities for composition:

- Combining two SQL queries to make a new SQL query
- Combining two snippets of HTML to make a new snippet of HTML
- Combining two shapes to make a new shape

17.3 Composing with identity: Monoids

Another type class that's similar to `Semigroup` is `Monoid`. The only major difference between `Semigroup` and `Monoid` is that `Monoid` requires an identity element for the type. An identity element means that `x <> id = x` (and `id <> x = x`). So for addition of integers, the identity element would be 0. But in its current state, your `Color` type doesn't have an identity element. Having an identity element might seem like a small detail, but it greatly increases the power of a type by allowing you to use a fold function to easily combine lists of the same type.

The `Monoid` type class is also interesting because it demonstrates an annoying problem in the evolution of Haskell type classes. Logically, you'd assume that the definition of `Monoid` would look like the following.

Listing 17.6 The rational definition of `Monoid`

```
class Semigroup a => Monoid a where
   identity :: a
```

QC 17.3 answer Yes, because addition of integers is associative: `1 + (2 + 3) = (1 + 2) + 3`.

After all, Monoid should be a subclass of Semigroup because it's just Semigroup with identity. But Monoid predates Semigroup and isn't officially a subclass of Semigroup. Instead, the definition of Monoid is perplexing.

Listing 17.7 The actual definition of Monoid

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Why mempty instead of identity? Why mappened instead of <>? These oddities in naming occur because the Monoid type class was added to Haskell before Semigroup. The most common Monoid is a list. The empty list is the identity for lists, and ++ (the append operator) is the <> operator for lists. The strange names of Monoid's methods are just m (for Monoid) tacked onto common list functions: empty, append, and concat. Here you can compare all three ways to do the same identity operation on a list:

```
GHCi> [1,2,3] ++ []
[1,2,3]
GHCi> [1,2,3] <> []
[1,2,3]
GHCi> [1,2,3] `mappend` mempty
[1,2,3]
```

Notice that mappend has the exact same type signature as <>.

Quick check 17.4 If you implement mappend/<> for Integer as * instead of +, what will your mempty value be?

17.3.1 mconcat: Combining multiple Monoids at once

The easiest way to see how powerful identity is, is to explore the final method in the definition of Monoid: mconcat. The only required definitions in Monoid are mempty and mappend.

QC 17.4 answer

1, because x × 1 = x.

If you implement these two, you get `mconcat` for free. If you look at the type signature of `mconcat`, you get a good sense of what it does:

```
mconcat :: Monoid a => [a] -> a
```

The `mconcat` method takes a list of `Monoid`s and combines them, returning a single `Monoid`. The best way to understand `mconcat` is by taking a list of lists and seeing what happens when you apply `mconcat`. To make things easier, you'll use strings because those are just lists of `Char`s:

```
GHCi> mconcat ["does"," this"," make"," sense?"]
"does this make sense?"
```

The great thing about `mconcat` is that because you've defined `mempty` and `mappend`, Haskell can automatically infer `mconcat`! This is because the definition of `mconcat` relies only on `foldr` (lesson 9), `mappend`, and `mempty`. Here's the definition of `mconcat`:

```
mconcat = foldr mappend mempty
```

Any type class method can have a default implementation, provided the implementation needs only a general definition.

17.3.2 Monoid laws

Just like `Semigroup`, there are `Monoid` type class laws. There are four:

- The first is that `mappend mempty x` is `x`. Remembering that `mappend` is the same as `(++)`, and `mempty` is `[]` for lists, this intuitively means that
  ```
  []  ++ [1,2,3] = [1,2,3]
  ```
- The second is just the first with the order reversed: `mappend x mempty` is `x`. In list form this is
  ```
  [1,2,3] ++ [] = [1,2,3]
  ```
- The third is that `mappend x (mappend y z) = mappend (mappend x y) z`. This is just associativity, and again for lists this seems rather obvious:
  ```
  [1] ++ ([2] ++ [3]) = ([1] ++ [2]) ++ [3]
  ```
  Because this is a `Semigroup` law, then if `mappend` is already implemented as `<>`, this law can be assumed because it's required by the `Semigroup` laws.
- The fourth is just our definition of `mconcat`:
  ```
  mconcat = foldr mappend mempty
  ```

Note that the reason `mconcat` uses `foldr` instead of `foldl` is due to the way that `foldr` can work with infinite lists, whereas `foldl` will force the evaluation.

17.3.3 Practical Monoids—building probability tables

Now let's look at a more practical problem you can solve with monoids. You'd like to create probability tables for events and have an easy way to combine them. You'll start by looking at a simple table for a coin toss. You have only two events: getting heads or getting tails. Table 17.1 is your table.

Table 17.1 Probability of heads or tails

Event	Probability
Heads	0.5
Tails	0.5

You have a list of `String`s representing events and a list of `Double`s representing probabilities.

Listing 17.8 Type synonyms for `Events` and `Probs`

```
type Events = [String]
type Probs = [Double]
```

Your probability table is just a list of events paired with a list of probabilities.

Listing 17.9 `PTable` data type

```
data PTable = PTable Events Probs
```

Next you need a function to create a `PTable`. This function will be a basic constructor, but it'll also ensure that your probabilities sum to 1. This is easily achieved by dividing all the probabilities by the sum of the probabilities.

Listing 17.10 `createPTable` makes a `PTable` ensuring all probabilities sum to 1

```
createPTable :: Events -> Probs -> PTable
createPTable events probs = PTable events normalizedProbs
 where totalProbs = sum probs
       normalizedProbs = map (\x -> x/totalProbs) probs
```

You don't want to get too far without making `PTable` an instance of the `Show` type class. First you should make a simple function that prints a single row in your table.

Listing 17.11 `showPair` creates a `String` for a single event-probability pair

```
showPair :: String -> Double -> String
showPair event prob = mconcat [event,"|", show prob,"\n"]
```

Notice that you're able to use `mconcat` to easily combine this list of strings. Previously, you used the `++` operator to combine strings. It turns out that `mconcat` not only requires less typing, but also provides a preferable way to combine strings. This is because there are other text types in Haskell (discussed in unit 4) that support `mconcat`, but not `++`.

To make `PTable` an instance of `Show`, all you have to do is use `zipWith` on your `showPair` function. This is the first time you've seen `zipWith`. This function works by zipping two lists together and applying a function to those lists. Here's an example adding two lists together:

```
GHCi> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
```

Now you can use `zipWith` to make your `PTable` an instance of `Show`.

Listing 17.12 Making `PTable` an instance of `Show`

```
instance Show PTable where
   show (PTable events probs) = mconcat pairs
     where pairs = zipWith showPair events probs
```

In GHCi, you can see that you have the basic setup you need:

```
GHCi> createPTable ["heads","tails"] [0.5,0.5]
heads|0.5
tails|0.5
```

What you want to be able to model using the `Monoid` type class is the combination of two (or more) `PTables`. For example, if you have two coins, you want an outcome like this:

```
heads-heads|0.25
heads-tails|0.25
tails-heads|0.25
tails-tails|0.25
```

This requires generating a combination of all events and all probabilities. This is called the *Cartesian product*. You'll start with a generic way to combine the Cartesian product of

two lists with a function. The `cartCombine` function takes three arguments: a function for combining the two lists, and two lists.

Listing 17.13 The `cartCombine` function for the Cartesian product of lists

```
cartCombine :: (a -> b -> c) -> [a] -> [b] -> [c]
cartCombine func l1 l2 = zipWith func newL1 cycledL2
 where nToAdd = length l2
       repeatedL1 = map (take nToAdd . repeat) l1
       newL1 = mconcat repeatedL1
       cycledL2 = cycle l2
```

You need to repeat each element in the first list once for each element in the second.

The preceding line leaves you with a lists of lists, and you need to join them.

Maps l1 and makes nToAdd copies of the element

By cycling the second list, you can use zipWith to combine these two lists.

Then your functions for combining events and combining probabilities are specific cases of `cartCombine`.

Listing 17.14 `combineEvents` and `combineProbs`

```
combineEvents :: Events -> Events -> Events
combineEvents e1 e2 = cartCombine combiner e1 e2
 where combiner = (\x y -> mconcat [x,"-",y])

combineProbs :: Probs -> Probs -> Probs
combineProbs p1 p2 = cartCombine (*) p1 p2
```

When combining events, you hyphenate the event names.

To combine probabilities, you multiply them.

With your `combineEvent` and `combineProbs`, you can now make `PTable` an instance of `Semigroup`.

Listing 17.15 Making `PTable` an instance of `Semigroup`

```
instance Semigroup PTable where
  (<>) ptable1 (PTable [] []) = ptable1
  (<>) (PTable [] []) ptable2 = ptable2
  (<>) (PTable e1 p1) (PTable e2 p2) = createPTable newEvents newProbs
    where newEvents = combineEvents e1 e2
          newProbs = combineProbs p1 p2
```

You want to handle the special case of having an empty PTable.

Finally, you can implement the `Monoid` type class. For this class, you know that `mappend` and `<>` are the same. All you need to do is determine the identity, `mempty` element. In this case, it's `PTable [] []`. Here's your instance of `Monoid` for `PTable`.

Listing 17.16 Making `PTable` an instance of `Monoid`

```
instance Monoid PTable where
   mempty = PTable [] []
   mappend = (<>)
```

Don't forget: you gain the power of `mconcat` for free!

To see how all this works, let's see how to create two `PTables`. The first is a fair coin, and the other is a color spinner with different probabilities for each spinner.

Listing 17.17 Example `PTables coin` and `spinner`

```
coin :: PTable
coin = createPTable ["heads","tails"] [0.5,0.5]

spinner :: PTable
spinner = createPTable ["red","blue","green"] [0.1,0.2,0.7]
```

If you want to know the probability of getting `tails` on the coin and `blue` on the spinner, you can use your `<>` operator:

```
GHCi> coin <> spinner
heads-red|5.0e-2
heads-blue|0.1
heads-green|0.35
tails-red|5.0e-2
tails-blue|0.1
tails-green|0.35
```

For your output, you can see that there's a 0.1, or 10%, probability of flipping `tails` and spinning `blue`.

What about the probability of flipping heads three times in a row? You can use `mconcat` to make this easier:

```
GHCi> mconcat [coin,coin,coin]
heads-heads-heads|0.125
heads-heads-tails|0.125
```

```
heads-tails-heads|0.125
heads-tails-tails|0.125
tails-heads-heads|0.125
tails-heads-tails|0.125
tails-tails-heads|0.125
tails-tails-tails|0.125
```

In this case, each outcome has the same probability: 12.5%.

Initially, the idea of abstracting out "combining things" might seem a bit too abstract. Once you start seeing problems in terms of monoids, it's remarkable how frequently they appear every day. Monoids are a great demonstration of the power of thinking in types when writing code.

Summary

In this lesson, our objective was to introduce you to two interesting type classes in Haskell: `Semigroup` and `Monoid`. Though both classes have rather strange names, they provide a relatively simple role. `Monoid` and `Semigroup` allow you to combine two instances of a type into a new instance. This idea of abstraction through composition is an important one in Haskell. The only difference between `Monoid` and `Semigroup` is that `Monoid` requires you to specify an identity element. `Monoid` and `Semigroup` are also a great introduction to the abstract thinking typically involved in more-advanced type classes. Here you start to see the philosophical difference between type classes in Haskell and interfaces in most OOP languages. Let's see if you got this.

Q17.1 Your current implementation of `Color` doesn't contain an identity element. Modify the code in this unit so that `Color` does have an identity element, and then make `Color` an instance of `Monoid`.

Q17.2 If your `Events` and `Probs` types were data types and not just synonyms, you could make them instances of `Semigroup` and `Monoid`, where `combineEvents` and `combineProbs` were the `<>` operator in each case. Refactor these types and make instances of `Semigroup` and `Monoid`.

22 LESSON

Lesson 22 from *Get Programming with Haskell* by Will Kurt

INTERACTING WITH THE COMMAND LINE AND LAZY I/O

After reading lesson 22, you'll be able to

- Access command-line arguments
- Use the traditional approach to interacting through I/O
- Write I/O code using lazy evaluation to make I/O easier

Often when people first learn about I/O and Haskell, they assume that I/O is somewhat of a challenge for Haskell because Haskell is all about pure programs and I/O is anything but pure. But there's another way to view I/O that makes it uniquely suited to Haskell, and somewhat clunky in other programming languages. Often when working with I/O in any language, we talk about *I/O streams*, but what is a stream? One good way to understand I/O streams is as a lazily evaluated list of characters. STDIN streams user input into a program until an eventual end is reached. But this end isn't always known (and in theory could never occur). This is exactly how to think about lists in Haskell when using lazy evaluation.

This view of I/O is used in nearly every programming language when reading from large files. Often it's impractical, or even impossible, to read a large file into memory before operating on it. But imagine that a given large file was simply some text assigned to a variable, and that variable was a lazy list. As you learned earlier, lazy evaluation

allows you to operate on infinitely long lists. No matter how large your input is, you can handle it if you treat the problem like a large list.

In this lesson, you'll look at a simple problem and solve it in a few ways. All you want to do is create a program that reads in an arbitrarily long list of numbers entered by a user, and then adds them all up and returns the result to the user. Along the way, you'll learn both how to write traditional I/O and how to use lazy evaluation to come up with a much easier way to reason about the solution.

Consider this You want to write a program that will let a user test whether words are palindromes. This is easy for a single word, but how can you let the user supply a continuous list of potential palindromes and keep checking as long as the user has words to check?

22.1 Interacting with the command line the nonlazy way

First let's design a command-line tool that reads a list of numbers entered by the user and adds them all up. You'll create a program called sum.hs. In the preceding lesson, you dealt with taking in user inputs and performing computations on them. The tricky thing this time is that you don't know how many items the user is going to enter in advance.

One way to solve this is to allow the user to enter a value as an argument to the program; for example:

```
$ ./sum 4
"enter your numbers"
 3
 5
 9
 25
"your total is 42"
```

To get arguments, you can use the `getArgs` function found in `System.Environment`. The type signature of `getArgs` is as follows:

```
getArgs :: IO [String]
```

So you get a list of `String`s in the context of `IO`. Here's an example of using `getArgs` in your `main`.

Listing 22.1 Getting command-line arguments by using `getArgs`

```
import System.Environment

main :: IO ()
main = do
 args <- getArgs
```

To get a feel for how `getArgs` works, it would be nice to print out all the `args` you have. Because you know that `args` is a list, you could use `map` to iterate over each value. But you have a problem, because you're working in the context of a `do` statement with an `IO` type. What you want is something like this.

Listing 22.2 Proposed solution to print your `args` (note: won't compile)

```
map putStrLn args
```

But `args` isn't an ordinary list, and `putStrLn` isn't an ordinary function. You can map over a list of values in `IO` with a special version of `map` that operates on `List`s in the context of `IO` (technically, on any member of the `Monad` type class). For that, there's a special helper function called `mapM` (the `M` stands for *Monad*).

Listing 22.3 Next improvement: using `mapM` (still won't compile)

```
main :: IO ()
main = do
 args <- getArgs
 mapM putStrLn args
```

Now when you compile your program, you still end up getting an error:

```
Couldn't match type '[()]' with '()'
```

GHC is complaining because the type of `main` is supposed to be `IO ()`, but you'll recall that `map` always returns a list. The trouble is that you just want to iterate over `args` and perform an `IO` action. You don't care about the results, and don't want a list back at the end. To solve this, there's another function called `mapM_` (note the underscore). This works just like `mapM` but throws away the results. Typically, when a function ends with an underscore in Haskell, it indicates that you're throwing away the results. With this small refactor, you're ready to go:

```
main :: IO ()
main = do
 args <- getArgs
 mapM_ putStrLn args
```

You can try a few commands and see what you get:

```
$ ./sum
$ ./sum 2
2
$ ./sum 2 3 4 5
2
3
4
5
```

Quick check 22.1 Write a `main` that uses `mapM` to call `getLine` three times, and then use `mapM_` to print out the values' input. (Hint: You'll need to throw away an argument when using `mapM` with `getLine`; use `(\_ -> ...)` to achieve this.)

Now you can add the logic to capture your argument. You should also cover the case of a user failing to enter an argument. You'll treat that as 0 lines. Also note that you're using the `print` function for the first time. The `print` function is `(putStrLn . show)` and makes printing any type of value easier.

Listing 22.4 Using a command-line argument to determine how many lines to read

```
main :: IO ()
main = do
 args <- getArgs
 let linesToRead = if length args > 0
                   then read (head args)
                   else 0 :: Int
 print linesToRead
```

QC 22.1 answer

```
exampleMain :: IO ()
exampleMain = do
   vals <- mapM (\_ -> getLine) [1..3]
   mapM_ putStrLn vals
```

Now that you know how many lines you need, you need to repeatedly call `getLine`. Haskell has another useful function for iterating in this way called `replicateM`. The `replicateM` function takes a value for the number of times you want to repeat and an `IO` action and repeats the action as expected. You need to import `Control.Monad` to do this.

Listing 22.5 Reading a number of lines equal to the user's argument

```
import Control.Monad

main :: IO ()
main = do
 args <- getArgs
 let linesToRead = if length args > 0
                   then read (head args)
                   else 0
 numbers <- replicateM linesToRead getLine
 print "sum goes here"
```

Okay, you're almost there! Remember that `getLine` returns a `String` in the `IO` context. Before you can take the sum of all these arguments, you need to convert them to `Int`s, and then you can return the sum of this list.

Listing 22.6 The full content of your sum.hs program

```
import System.Environment
import Control.Monad

main :: IO ()
main = do
 args <- getArgs
 let linesToRead = if length args > 0
                   then read (head args)
                   else 0 :: Int
 numbers <- replicateM linesToRead getLine
 let ints = map read numbers :: [Int]
 print (sum ints)
```

That was a bit of work, but now you have a tool that lets users enter as many `int`s as they want, and you can add them up for them:

```
$ ./sum 2
4
59
$ ./sum 4
1
2
3
410
```

Even in this simple program, you've covered a number of the tools used to handle user inputs. Table 22.1 covers some useful functions for iterating in an `IO` type.

Table 22.1 Functions for iterating in an IO context

Function	Behavior
`mapM`	Takes an `IO` action and a regular list, performing the action on each item in the list, and returning a list in the `IO` context
`mapM_`	Same as `mapM`, but it throws away the values (note the underscore)
`replicateM`	Takes an `IO` action, an `Int` *n*, and then repeats the `IO` action *n* times, returning the results in an `IO` list
`replicateM_`	Same as `replicateM`, but it throws away the results

Next you'll look at how much easier this would be if you used lazy evaluation.

Quick check 22.2 Write your own version of `replicateM`, `myReplicateM`, that uses `mapM`. (Don't worry too much about the type signature.)

22.2 Interacting with lazy I/O

Your last program worked but had a few issues. First is that you require the user to input the specific number of lines needed. The user of your sum program needs to know this ahead of time. What if users are keeping a running tally of visitors to a museum, or

QC 22.2 answer

```
myReplicateM :: Monad m => Int -> m a -> m [a]
myReplicateM n func = mapM (\_ -> func) [1 .. n]
```

piping in the output of another program to yours? Recall that the primary purpose of having an `IO` type is to separate functions that absolutely must work in I/O with more general ones. Ideally, you want as much of your program logic outside your `main`. In this program, all your logic is wrapped up in `IO`, which indicates that you're not doing a good job of abstracting out your overall program. This is partially because so much I/O behavior is intermingled with what your program is supposed to be doing.

The root cause of this issue is that you're treating your I/O data as a sequence of values that you have to deal with immediately. An alternative is to think of the stream of data coming from the user in the same way you would any other list in Haskell. Rather than think of each piece of data as a discrete user interaction, you can treat the entire interaction as a list of characters coming from the user. If you treat your input as a list of `Chars`, it's much easier to design your program and forget all about the messy parts of I/O. To do this, you need just one special action: `getContents`. The `getContents` action lets you treat the I/O stream for STDIN as a list of characters.

You can use `getContents` with `mapM_` to see how strangely this can act. You'll be working with a new file named sum_lazy.hs for this section.

Listing 22.7 A simple `main` to explore lazy I/O

```
main :: IO ()
main = do
  userInput <- getContents
  mapM_ print userInput
```

The `getContents` action reads input until it gets an end-of-file signal. For a normal text file, this is the end of the file, but for user input you have to manually enter it (usually via Ctrl-D in most terminals). Before running this program, it's worth thinking about what's going to happen, given lazy evaluation. In a strict (nonlazy) language, you'd assume that you have to wait until you manually enter Ctrl-D before your input would be printed back to use. Let's see what happens in Haskell:

```
$ ./sum_lazy
hi
'h'
'i'
'\n'
what?
'w'
```

```
'h'
'a'
't'
'?'
'\n'
```

As you can see, because Haskell can handle lazy lists, it's able to process your text as soon as you enter it! This means you can handle continuous interaction in interesting ways.

Quick check 22.3 Use lazy I/O to write a program that reverses your input and prints it back to you.

22.2.1 Thinking of your problem as a lazy list

With `getContents`, you can rewrite your program, this time completely ignoring `IO` until later. All you need to do now is take a list of characters consisting of numbers and newline characters `\n`. Here's a sample list.

Listing 22.8 Sample data representing a string of input characters

```
sampleData = ['6','2','\n','2','1','\n']
```

If you can write a function that converts this into a list of `Int`s, you'll be all set! There's a useful function for `String`s that you can use to make this easy. The `lines` function allows you to split a string by lines. Here's an example in GHCi with your sample data:

```
GHCi> lines sampleData
["62","21"]
```

The `Data.List.Split` module contains a more generic function than `lines`, `splitOn`, which splits a `String` based on another `String`. `Data.List.Split` isn't part of base Haskell, but is

QC 22.3 answer

```
reverser :: IO ()
reverser = do
   input <- getContents
   let reversed = reverse input
   putStrLn reversed
```

included in the Haskell Platform. If you aren't using the Haskell Platform, you may need to install it. The `splitOn` function is a useful one to know when processing text. Here's how lines could be written with `splitOn`.

Listing 22.9 Defining `myLines` with `splitOn` from `Data.List.Split`

```
myLines = splitOn "\n"
```

With lines, all you need is to map the `read` function over your new lists and you'll get your list of `Int`s. You'll create a `toInts` function to do this.

Listing 22.10 `toInts` function to convert your `Char` list into a list of `Int`s

```
toInts :: String -> [Int]
toInts = map read . lines
```

Making this function work with `IO` is remarkably easy. You apply it to your `userInput` you captured with `getContents`.

Listing 22.11 Your lazy solution to processing your numbers

```
main :: IO ()
main = do
  userInput <- getContents
  let numbers = toInts userInput
  print (sum numbers)
```

As you can see, your final `main` is much cleaner than your first version. Now you can compile your program and test it out:

```
$ ./sum_lazy
4
234
23
1
3
<ctrl-d>
265
```

This is much nicer than before, as your code is cleaner and users don't have to worry about how many numbers are in the list when they start. In this lesson, you've seen how

to structure your program to work in a way similar to most other programming languages. You request data from the user, process that data, and then request more input from the user. In this model, you're performing *strict* I/O, meaning that you evaluate each piece of data as you get it. In many cases, if you treat the user input as a regular lazy list of `Chars`, you can abstract out nearly all of your non-I/O code much more easily. In the end, you have only one point where you need to treat your list as I/O: when you first receive it. This allows all the rest of your code to be written as code that operates on a normal list in Haskell.

Quick check 22.4 Write a program that returns the sum of the squares of the input.

Summary

In this lesson, our objective was to introduce you to the ways to write simple command-line interfaces in Haskell. The most familiar way is to treat I/O just like any other programming language. You can use do-notation to create a procedural list of `IO` actions, and build interactions with I/O this way. A more interesting approach, possible in few languages other than Haskell, is to take advantage of lazy evaluation. With lazy evaluation, you can think of the entire input stream as a lazily evaluated list of characters, `[Char]`. You can radically simplify your code by writing out pure functions as though they were just working on the type `[Char]`. Let's see if you got this.

Q22.1 Write a program, simple_calc.hs, that reads simple equations involving adding two numbers or multiplying two numbers. The program should solve the equation each user types into each line as each line is entered.

Q22.2 Write a program that allows a user to select a number between 1 and 5 and then prints a famous quote (quotes are of your choosing). After printing the quote, the program will ask whether the user would like another. If the user enters `n`, the program ends; otherwise, the user gets another quote. The program repeats until the user enters `n`. Try to use lazy evaluation and treat the user input as a list rather than recursively calling `main` at the end.

QC 22.4 answer

```
mainSumSquares :: IO ()
mainSumSquares = do
   userInput <- getContents
   let numbers = toInts userInput
   let squares = map (^2) numbers
   print (sum squares)
```

Chapter 3 from *Haskell in Depth*
by Vitaly bragilevsky

Processing stock quote data: An example

This chapter covers

- Designing a standalone multi-module Haskell program with external packages
- Dealing with dates, times, texts, and command-line arguments
- Parsing CSV files and drawing charts and
- Employing type classes for practical needs

A common pattern for many utility programs is that: you have a file with data in a form which isn't convenient for analysis, and your goal's to present this data, either graphically or textually. When implementing such a program you'll have to address many issues, such as interfacing with the user, designing datatypes for the application domain, reusing external packages for parts of the program, and more. You should also think about how language features can help you in terms of correctness, efficiency, and the ability to extend functionality.

In this chapter we'll explore the process of developing such a program. I'll start by describing inputs and outputs, then move on to design issues with datatypes and functions, followed by a discussion of useful packages and implementation details. You'll also see how type classes can make your programs much more flexible and resilient to changes.

3.1 Setting the scene

The task is as follows: take the historical quote data for some joint-stock company in CSV format (a text file with comma-separated values) and turn it into a statistical report (as textual information) and several charts (graphical information). Figure 3.1 presents the overall data flow of the resulting program: you need to read the CSV file into a collection of `QuoteData` values and then process this collection in order to gather statistical information and draw charts.

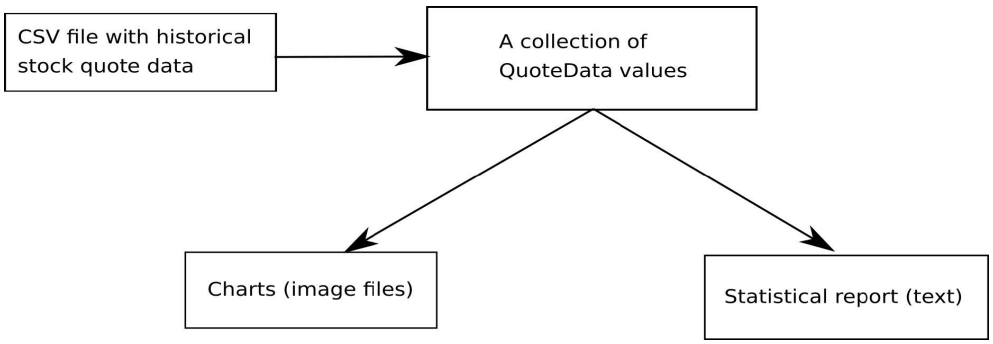


Figure 3.1 The example's goal: producing charts and stats from a CSV file with quote data

Here's a sample of the data file, `data/quotes.csv`:

```
day,close,volume,open,high,low
2017/10/11,156.5500,16861450.0000,155.9700,156.9800,155.7500
2017/10/10,155.9000,15603520.0000,156.0550,158.0000,155.1000
2017/10/09,155.8400,16243080.0000,155.8100,156.7300,155.4850
2017/10/06,155.3000,17223790.0000,154.9700,155.4900,154.5600
2017/10/05,155.3900,21215870.0000,154.1800,155.4400,154.0500
...
```

The first line lists the six fields, and every other line of this file contains their values:

- `day` is the date of the stock transactions.
- `close` is the share price at the close of business.
- `volume` is the total number of shares of the stock traded during the day.
- `open` is the price at the opening.
- `high` is the highest price during the day.
- `low` is the lowest price during the day.

I'm not going to teach you about financial analysis or trend prediction. Instead, you'll draw two charts, one for prices and another for volume (see figure 3.2 for sample charts).

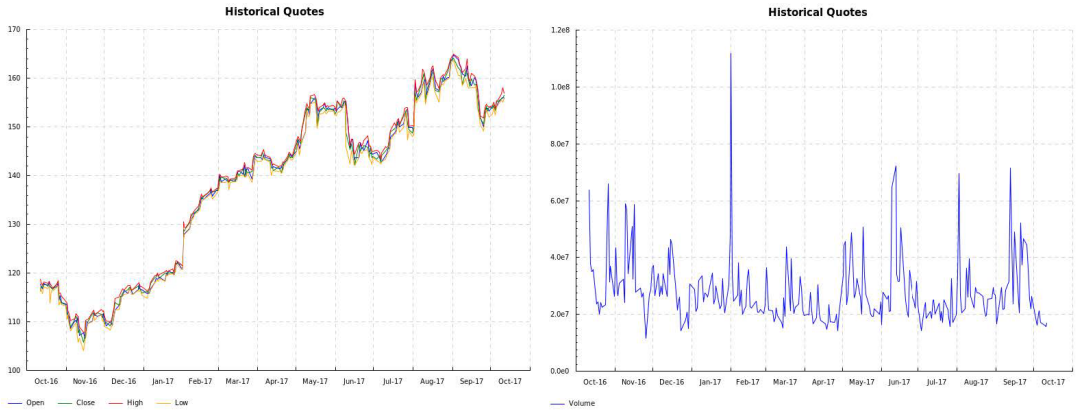


Figure 3.2 Charts of stock quotes for some company

You'll also compute simple characteristics of the data, such as the mean, minimal, and maximal values of the fields, and the number of days between the minimal and maximal values being reached for the given period (and yes, I call this "a statistical report"):

```
Statistics for Open:
  Mean: 138.0890
  Min: 106.5700
  Max: 164.8000
  Days between Min/Max: 290
...
```

See figure 3.3 for a complete picture of what this program does.

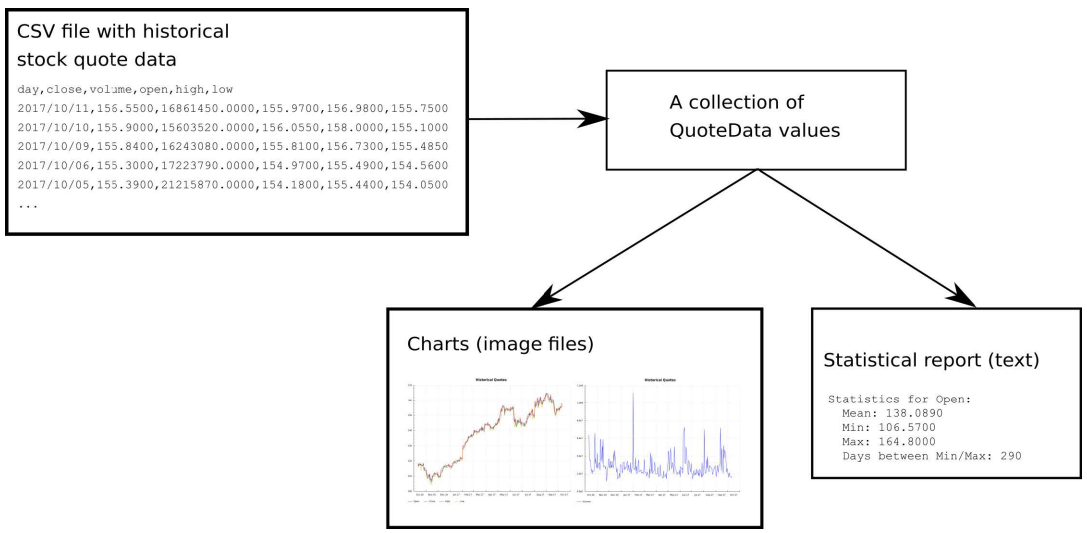


Figure 3.3 The goal: processing stock quotes

As you write this program we'll discuss and solve many common practical problems, including:

- Representing data—We have dates and volumes/currency values with fixed precision.
- Parsing CSV files—We can either employ an ad hoc solution or deal with it a bit more professionally.
- Computing statistics—Our main goal's to reuse code as much as possible, and we'll try to use higher-order functions for this.
- Building a statistical report—A report's a text with data structured in some way; this should be addressed with flexibility and extensibility in mind.
- Drawing charts—We want to use sophisticated packages.
- Designing UI—We should deal with command-line arguments, prepare to see `Semigroup` and `Applicative` in action. Are you astonished?
- Maintaining a clear division between pure and I/O parts of the program—We'll aim to keep the latter as small as possible.

This is still a study example, and we'll leave out performance, exception handling, testing, and many other issues for now.

3.2 Exploring design space

The first thing to do is define types to represent the data. Let's call one piece of data `QuoteData`—this is a record with all the fields for day, prices, and volume. Here comes the problem: how do you represent dates and prices? Is it a good idea to use `String` and `Double`? Definitely not. Before anything else, let's look at some packages that can help us with this. We'll then move to other issues that should be dealt with.

3.2.1 Dealing with dates and times

Many factors should be taken into account when using dates and times in your programs. First, you must decide which calendar to use. These days the most straightforward solution's to stick with the Gregorian calendar, but there are other options. Processing dates in the first millennium AD requires using the Julian calendar, although writing software for businesses or governments with fiscal years in mind might result in employing the ISO week date system (as defined in ISO 8601). Referring to time means dealing with timestamps, moments in time with respect to time zones, or durations. Time zones introduce the issue of Daylight Savings Time. And it gets worse: what about *leap seconds*, which are irregularly added to the year due to the changes in the Earth's rate of rotation around the Sun?

Fortunately, the `time` package in Haskell is sophisticated enough to deal with all these technical difficulties, and using the type system to prevent its users from making mistakes in mixing times and dates. Table 3.2 contains short descriptions of the most useful types. All of them are accessible after installing the `time` package and importing the module `Data.Time`.

Table 3.1 Types for representing dates and times in the `time` package

Type	Description
`Day`	The date in the Gregorian calendar (which is stored as a count of days, with zero being the day November 17, 1858)
`TimeOfDay`	The time of day as represented in hour, minute, and second (with picoseconds)
`LocalTime`	A simple aggregation of `Day` and `TimeOfDay` that can be used to represent local times without referring to time zones, as in the phrase 'office opens on January 3, 2019, at 9am'
`TimeZone`	A time zone offset like `-0700` (which isn't the name of a time zone—refer to the package `tz` for actual time zone processing)
`ZonedTime`	An aggregation of a `LocalTime` with a `TimeZone`
`UTCTime`	The time as a UTC timestamp, which consists of the day number and a time offset from midnight, regardless of location
`NominalDiffTime`	Durations as differences between times
`SystemTime`	Time as measured by the system's clock, which can be used for low-latency timing

Apart from these datatypes the package `time` provides many functions, including functions for:

- Constructing dates and times from integer values (like years, months, days, hours, minutes, and seconds)
- Parsing dates and times from strings (with the ability to specify the actual format)
- Formatting dates and times into strings (by specified formats and with somewhat limited localization)
- Getting the current date and time (which requires `IO`)
- Manipulating dates and times, such as by adding date intervals or computing differences

You can find details about this package in the documentation on Hackage (https://hackage.haskell.org/package/time) or in this tutorial: https://two-wrongs.com/haskell-time-library-tutorial.

It should be clear that the best choice for your program's the `Day` datatype, because you don't have times in your CSV files and are only interested in recent stock quote data.

3.2.2 Computing with fixed precision

Looking at the original data reveals that share prices and volumes are given as real numbers with fixed precision (four decimal places). This is often the case with financial computations: for example, share prices in US dollars can keep two more decimal places after cents to avoid being affected too much by rounding errors.

If you want to keep the same precision and at the same time compute minimal, maximal, and mean values, then you should use a specific type. In Haskell you can use the `Data.Fixed` module from `base`, but unfortunately it only defines types with 0, 1, 2, 3, 6, 9, and 12 decimal places out of the box, and you'll have to define your own type for 4 decimal places.

The module `Data.Fixed` suggests the following path to get such a type. First, you need a type that allows the compiler to choose the right resolution for your numeric type (which is ten to the power of the number of digits after the decimal point you're interested in, and `10000` in this case, as you need four digits). For specifying such a resolution, the module `Data.Fixed` provides the type class `HasResolution`, which has one method, `resolution`:

```
class HasResolution a where
  resolution :: p a -> Integer
  {-# MINIMAL resolution #-}
```

You need a type and an instance of the `HasResolution` type class for it. Because there are no specific requirements for that type, apart from having such an instance, you're free to use an *empty datatype*.

```
data E4

instance HasResolution E4 where
    resolution _ = 10000
```

The only role the type `E4` is playing is choosing the right instance, and you don't need any data in there.

The numeric type with the specific number of decimal places is defined with a new type, `Fixed`, parameterized by `E4` (this is why you have the type signature `p a` in the definition of `resolution`: `p` refers to the new type `Fixed` and `a` to its parameter):

```
type Fixed4 = Fixed E4
```

Now you can be sure that this particular implementation of the `resolution` method will be applied to the `Fixed4` values in order to limit the number of digits after the decimal point:

```
ghci> pi = 3.14 :: Fixed4
ghci> pi
3.1400
ghci> resolution pi
10000
ghci> e = 2.7182818 :: Fixed4
ghci> e
2.7182
ghci> resolution e
10000
```

The type `Fixed4` doesn't round the value but truncates it instead, that's why `e` constant is shown here as `2.7182` and not `2.7183`.

In the following GHCi session you can see how this type can be used in computations:

```
ghci> pi * e
8.5351
```

Four decimal places are kept, thanks to the instances of various numeric type classes provided by the `Data.Fixed` module and the `HasResolution` type class itself.

Now you can use the `Fixed4` type for storing share prices from the given CSV file.

3.2.3 Parsing data

After considering the possible ways to represent dates, times, and numbers with fixed precision, we're ready to declare the `QuoteData` datatype:

```
data QuoteData = QuoteData { day :: Day, close :: Fixed4, volume :: Fixed4,
                             open :: Fixed4, high :: Fixed4, low :: Fixed4 }
```

Now that we've the datatype `QuoteData` for representing stock quote information for one day, you can represent all the data available as a list, `[QuoteData]` (which isn't extremely efficient but it'll suffice for now).

Parsing data files is a well known programming task. Either general solutions for writing virtually any parsers or libraries for parsing specific file formats can be used. It's much better to find a package which is able to consider all the corner cases, deal with errors, and address issues of performance and convenience.

The simplest solution without any external tools is the following:

- Split the file content into the lines.
- Skip the first line with the names of the fields.
- Transform all other lines values of `QuoteData` type.

Transforming file lines into `QuoteData` is the most challenging here: we should split the line into components, parse date from the first component and turn the others into `Fixed4` values, then we should create `QuoteData` value from the extracted components, as done in the following listing.

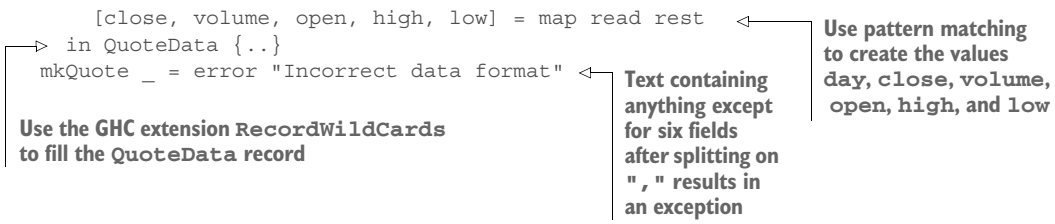
Listing 3.1 Parsing a CSV file content: ad hoc solution

```
text2Quotes :: T.Text -> [QuoteData]
text2Quotes = map (mkQuote . toComponents) . tail . T.lines
  where
    toComponents = map T.unpack . T.splitOn ","
    mkQuote (d : rest@[_,_,_,_,_]) =
      let
        day = parseTimeOrError False defaultTimeLocale "%Y/%m/%d" d
```

The composition of functions represents processing steps

Parse the given date using the function `parseTimeOrError` from the `time` package

Deconstruct and check the given list of `String` values



Note how advanced pattern matching's used in the definition of the `mkQuote` function in the previous listing: this function takes a list of `String` values from `toComponents` over single CSV file line, checks whether there are exactly six components, and combines all of the fields except the first one into the list `rest` for future processing. All this stuff in one line of code!

RecordWildCards GHC extensions

GHC extension `RecordWildCards` allows you to bring all the record fields into scope without mentioning their names explicitly. You can use it both for accessing their values in pattern matching and constructing records. For example, if you're given the `QuoteData` you can define function over it as follows:

```
isRising :: QuoteData -> Bool
isRising QuoteData {..} = close > open
```

Or, as in the `text2Quotes` example, you can construct a record:

```
zeroQD :: Day -> QuoteData
zeroQD d = let day = d
               close = 0
               open = 0
               high = 0
               low = 0
               volume = 0
           in QuoteData { .. }
```

Remember your code should have a `LANGUAGE` pragma mentioning the `RecordWildCards` extension in the beginning of the file in order to use it.

This first implementation uses the plain old strategy "garbage in, garbage out": any formatting errors in the original file, such as the wrong number of fields, the wrong date format, or NaN (not-a-number) values in other fields results in an exception, leading to the program halting. We could implement other approaches, like:

- Ignore incorrect lines silently or with reporting to the user
- Interpolate missing values somehow using neighboring values
- Stop reading the file after encountering an error

But they'd make the implementation a bit harder. Instead, we'll follow another path: we'll use an external package, `cassava`, designed specifically for parsing CSV files. The `cassava` package allows you to throw away hand-rolled CSV parsing and replace it with a carefully crafted, highly efficient implementation. But there's a price:

- You must describe your data in terms of this package, to define how to convert textual file content into stock quote data fields. This can be done by implementing instances of the `FromField` type class that comes with the `cassava` package.
- You must start working with `Vector` (found in the `Data.Vector` module), the data structure which is promoted by the package as the result of parsing. The good news is that this is almost transparent for us, thanks to using the `Foldable` type class, as a container constraint.

We'll return to the actual implementation with `cassava` shortly.

3.2.4 Formatting texts

Another common task when writing applications is combining text and data (see figure 3.4).

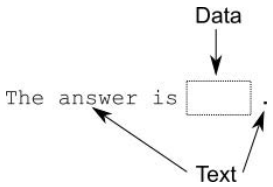


Figure 3.4 The problem of combining text and data

This can be done easily by concatenating strings and data converted to `String` values with `show`:

```
ghci> let a = 42 in "The answer is " ++ show a ++ "."
"The answer is 42."
```

But such a solution isn't always appropriate. It's bad in terms of performance and flexibility. The source of the performance drawbacks isn't the `String` type—replacing it with `Text` wouldn't solve the problem. The key idea here's to employ a *builder*, which is responsible for adding parts of text incrementally, one by one, and then concatenating all of them into one `String` or `Text` value. The builder should be able to absorb data of various types without converting them to `String` or `Text` explicitly. It should also support patterns to specify formats and rules for representing the data in textual form. The task of *formatting* text is close to *templating*, where you provide a template of the text with some tokens which are substituted by values at the final stage of processing.

You should consider several alternatives to do text formatting:

- `Data.Text.Lazy.Builder` from the `text` package provides a limited builder without any formatting; you can use it when you have many `Text` values that should be combined into one.
- The function `printf` from the `Text.Printf` module provides many format specifiers; it takes a format string and data and produces a `String`.
- The packages `text-format`, `formatting`, and the more recent `fmt` are close in their goals but different in their interfaces and implementation.
- The package `template` provides a simple template substitution engine, and other full-blown templating systems used mainly in web programming.

In this project we'll use the `fmt` package, but the same can be easily done with other text formatting packages as well. After enabling the `OverloadedStrings` GHC extension and importing the module `Fmt`, you can construct the formatting string by operators and then use the function `fmt` to create resulting `Text` value:

```
ghci> :set -XOverloadedStrings
ghci> import Fmt
ghci> name = "John"
ghci> age = 30
ghci> fmt $ "Hello, "+|name|+"!\nI know that your age is "+|age|+".\n"
Hello, John!
I know that your age is 30.
ghci> fmt $ "Which is "+|hexF age|+" in hex!\n"
Which is 1e in hex!
```

The operators `+|` and `|+` are used for including variables and formatters (which are ordinary functions like `hexF` here). Sometimes you need to call `show` for your variable (if the `fmt` package doesn't know how to convert it to textual form); this can be done implicitly via the other pair of operators, `+||` and `||+`. Formatters from the package `fmt` are powerful enough to present tuples, lists, and even associated lists, but you can always provide your own formatter for your data by writing a function returning `Builder`, which is a datatype used for efficiently constructing `Text` values.

The package `fmt` supports also another approach. Instead of writing functions we can specify how to *build* text from the data we have. This can be done using the type class `Buildable` from the `text-format` package, which is also used by the `fmt` package itself:

```
class Buildable p where
  build :: p -> Builder
```

The method `build` should turn every `Buildable` value into a `Builder`. We can provide such instances for our data and then use them while formatting the statistical report.

3.2.5 Other tasks and the corresponding packages

We still have two problems to address:

- How to draw charts for share prices and volumes
- How to develop the user interface

Let's discuss them briefly now.

DRAWING CHARTS

Unfortunately, Haskell isn't the best language for presenting data in visual form. Such languages as Python and R provide much better infrastructure and tooling. Nevertheless, we can still draw 2D charts and plots in Haskell, in this project we'll use the `Chart` package for that. It's a good example of a package built on top of other sophisticated Haskell packages, and it's instructive to discuss its ideas and implementation. The package `Chart` requires a *backend* for generating actual files, in our case we'll use `Chart-diagrams` to create SVG files. You can find many examples on the package wiki page on GitHub: https://github.com/timbod7/haskell-chart/wiki.

USER INTERFACE

For simplicity we'll stick with a non-interactive command-line interface. How about the following:

```
$ stockquotes -h
Usage: stockquotes FILE [-c|--company ARG] [-p|--prices] [-v|--volumes]
  Stock quotes data processing

Available options:
  FILE                     csv-file name
  -c,--company ARG         stock company's name
  -p,--prices              create file with prices chart
  -v,--volumes             create file with volumes chart
  -h,--help                Show this help text
```

This is a rather standard way for describing command-line interfaces. Imagine if it was generated automatically: programmers should describe arguments declaratively and everything else including parsing command-line arguments, checking them for errors, preparing configuration was generated by the package `optparse-applicative` out if that description.

Its idea is to force distinction between command-line arguments and the datatype for storing configuration parameters: we describe our options by specifying how they correspond to the configuration.

3.2.6 External packages in the project

I've already mentioned several packages apart from `base` that we'll need for this project. In fact, we'll need two more:

- `bytestring` for reading CSV file efficiently (`cassava` expects data in the form of a byte string)
- `safe` package provides many *safe* alternatives for partial functions from `Prelude`, we'll use them from time to time in this book.

All these packages are listed in the table 3.2. You should install them manually or they'll be installed for you if you follow an advice on working with source code examples. Technically when installing these packages several dozens of others will be installed for you as well as their dependencies.

Table 3.2 Used packages

Package	Used for
`text`	Efficient text processing and input/output
`time`	Dealing with dates and times
`fmt`	Formatting texts
`Chart, Chart-diagrams`	Drawing charts
`cassava`	Parsing CSV files
`optparse-applicative`	Processing command-line arguments
`bytestring`	Efficient Input/Output for binary data
`safe`	Safe alternatives to partial functions from `Prelude`

It's almost impossible to write useful programs without referring to external packages. We'll meet many other packages later in this book that you can use in your own projects.

3.3 Implementing stockquotes project

Let's think what should we do in this project and in what order:

- process command-line arguments
- read quotes data from CSV file
- compute statistics
- prepare textual report on statistical info
- plot charts

Depending on the supplied arguments some of these stages may be skipped.

It's a good practice in general to split the required functionality over the several modules, for example:

- `Params` for describing command-line arguments and processing them
- `QuoteData` for describing main datatypes in a form suitable for reading
- `Statistics` for computing statistical info
- `StatReport` for preparing report in a text form
- `Charts` for plotting charts

I've splitted preparing statistical report into two modules, `Statistics` and `StatReport` because it'd be useful whenever we need to prepare reports in other formats as well. Surely, we also need the `Main` module to connect the program components to each over and drive the whole program. Figure 3.5 demonstrates the module structure for this program with arrows pointing to the imported modules. This diagram was created with the help of the `graphmod` utility from Hackage, developed by Iavor S. Diatchki.

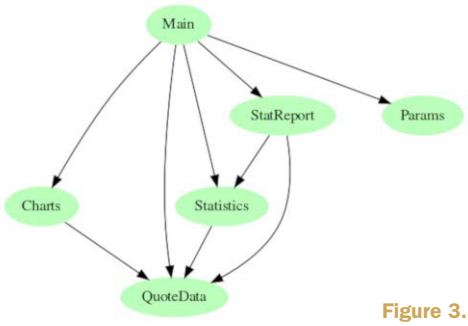


Figure 3.5 Module structure for stock quote data example

We're ready describe the program functionality with types and functions. First, we'll need types for the command-line parameters, the single quote data and collection, and the statistical info:

```
data Params
data QuoteData
data QuoteDataCollection
data StatInfo
```

We'll postpone the definition of these datatypes until we've enough information on what exactly should be in there.

The program should start by reading user input in the form of command-line arguments (normally a list of `String`) and then either do its job or inform the user about the correct way to run it. As we've decided to delegate this job to the `opt-parse-applicative` package we can imagine to have already specified `Params` and define:

```
work :: Params -> IO ()
```

There we'll have to read the stock quotes data from the CSV file:

```
readQuotes :: FilePath -> IO QuoteDataCollection
```

Compute the statistical information (purely!):

```
statInfo :: QuoteDataCollection -> StatInfo
```

Prepare the report (again, purely!):

```
statReport :: StatInfo -> Text
```

The simplest way to plot charts is to generate files with them; we'll have to stick with `IO` for this task:

```
plotCharts :: Params -> QuoteDataCollection -> IO ()
```

Figure 3.6 is an informal flowchart which presents the proposed structure of the program. You can see user input and both the I/O and pure parts of the program.

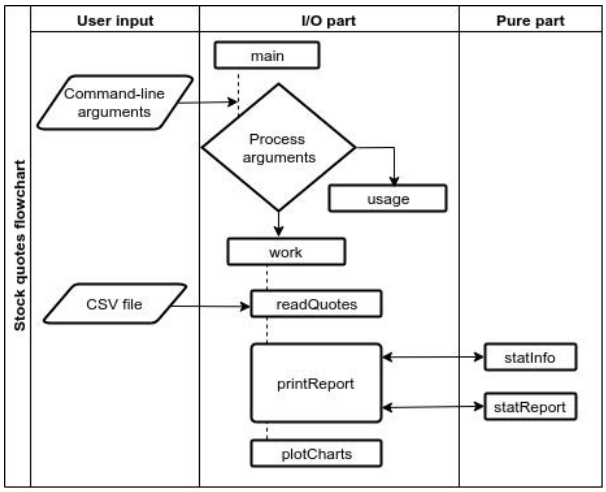


Figure 3.6 Processing stock quote data: program structure flowchart

We'll refine the types and names of these functions later, but even now they clearly represent the program functionality.

3.3.1 Describing data

Remember that our data's given in the form of CSV file `data/quotes.csv`:

```
day,close,volume,open,high,low
2017/10/11,156.5500,16861450.0000,155.9700,156.9800,155.7500
2017/10/10,155.9000,15603520.0000,156.0550,158.0000,155.1000
2017/10/09,155.8400,16243080.0000,155.8100,156.7300,155.4850
2017/10/06,155.3000,17223790.0000,154.9700,155.4900,154.5600
2017/10/05,155.3900,21215870.0000,154.1800,155.4400,154.0500
...
```

This is a CSV file with *named* fields. We've six fields in every line: the first represents the date and all the others are given as floating point numbers with fixed precision. We already know that we need the type `Day` from the `Data.Time` module (package `time`) and the type `Fixed4` based on `Data.Fixed` module (`base` package).

To describe our data in a form suitable for the `cassava` package we need to derive or define instances of several type classes, namely:

- `Generic` from the `GHC.Generics` module to give `cassava` tools for working with our datatypes using generic programming machinery (we'll discuss it later in this book in greater details);
- `FromNamedRecord` from the `Data.CSV` module to allow `cassava` to read CSV file with named fields for us: this is possible thanks to the same names in the CSV file and the `QuoteData` datatype;
- `FromField` from the `Data.CSV` module to teach `cassava` how to parse `Day` and `Fixed4` values, `cassava` is able to parse values of many types from `base` but it doesn't know how to deal with our types.

Note the heavy use of type classes as it is quite common in Haskell: the library can't imagine all types it's used with (those types may not even exist yet). Instead it describes their constraints and behaviour with type classes. Now it's our responsibility to provide the corresponding instances in order to use the library. Type class instances build a bridge between library's interface (API in the form of type classes and functions that rely on them) and the client's datatypes. We'll see the same idea at work many times later in this book.

The following listing presents the header of the module `QuoteData` with the required GHC extensions and imported modules. Note how we limit imports: we avoid introducing unnecessary names from the imported modules by specifying what we need. The syntax `(..)` in the import lists refers to everything inside, for example `HasResolution (..)` in the import list for `Data.Fixed` module means the type class `HasResolution` and every method of this type class (`resolution` in this case). We can use the same syntax for algebraic datatypes and their value constructors. Alternatively, we can list names of methods and value constructors explicitly if we want to import only some of them.

Listing 3.2 QuoteData module: extensions and imports

These extensions are required for deriving instances.

GHC extension `FlexibleInstances` allows to define instances for type synonyms and weakens the instance requirements from the Haskell Report in many other ways.

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

module QuoteData where
import Data.Fixed (HasResolution (..), Fixed)
import Data.Time (Day, parseTimeM, defaultTimeLocale)
import Safe (readDef)
import Data.ByteString.Char8 (unpack)
import GHC.Generics (Generic)
import Data.Csv (FromNamedRecord, FromField (..))
```

We use these types for defining the type for numbers with fixed precision.

We need this for defining and parsing the `Day` field.

Function `readDef` is a safe alternative for `read` which returns a default value in

Function `unpack` is used for getting a `String` to parse with `time` package machinery.

Now we can define the type `Fixed4` and provide the definition for `QuoteData` with the derived instances required by `cassava`:

```
data E4

instance HasResolution E4 where
    resolution _ = 10000

type Fixed4 = Fixed E4

data QuoteData = QuoteData {
                   day :: Day, close :: Fixed4, volume :: Fixed4,
                   open :: Fixed4, high :: Fixed4, low :: Fixed4
                 }
  deriving (Generic, FromNamedRecord)
```

The deriving machinery needs two instances of the `FromField` type class when trying to derive `FromNamedRecord` for `QuoteData`: one for `Day` and one for `Fixed4`. `Cassava` knows nothing about them and we need to provide them by ourselves. The type class `FromField` is defined in `Data.CSV` as follows:

```
class FromField a where
  parseField :: Field -> Parser a
  {-# MINIMAL parseField #-}
```

This type class defines how to parse field of type `a`. The type `Field` is, in fact, a synonym for `ByteString` (the reason we'll need to use `unpack`) and `Parser` is a monad for parsing used inside `cassava`. We already know the monad interface and we don't need to think about what is this `Parser` about—it's a monad, which is enough. The following listing gives all the details:

Listing 3.3 QuoteData module: providing instances for parsing values of `Day` and `Fixed4` types

```
instance FromField Fixed4 where
  parseField s = pure $ readDef 0 $ unpack s

instance FromField Day where
  parseField s = parseTimeM False defaultTimeLocale "%Y/%m/%d" (unpack s)
```

We take `Bytestring s`, unpack it into the `String` and then read it as a `Fixed4` value using `0` by default. The result's then fed to the monad `Parser` with `pure`.

We use `parseTimeM` that can work in any monad to parse a `Day` value, it reports a failure to the underlying monad in case of errors.

Many Haskell programmers prefer a shorter implementation for parseField that doesn't refer to the `s` variable, namely:

```
instance FromField Fixed4 where
  parseField = pure . readDef 0 . unpack
```

Such a definition's better as it presents parsing steps more clearly:

- unpacking
- reading
- returning to `Monad`

Haskellers use fancy name *eta-reduction* (or even *?-reduction* in Greek) which comes from lambda-calculus for rewriting function definitions from the first form to the second one.

Finally let's introduce a simple helper to work with `QuoteData` numeric fields more uniformly. It becomes convenient later when computing statistics: clearly there should be no difference in computing minimums or maximums for open or close share prices.

```
data QField = Open | Close | High | Low | Volume
  deriving (Show, Enum, Bounded, BoundedEnum)

field2fun :: QField -> QuoteData -> Fixed4
field2fun Open = open
field2fun Close = close
field2fun High = high
field2fun Low = low
field2fun Volume = volume
```

We've defined a value constructor for every numeric field in `QuoteData` and mapped it to the record fields (which are technically accessor functions `QuoteData → Fixed4`). As a result, we'll be able to write something like `field2fun qf q` to access any required field `qf` from a value `q` of type `QuoteData`. We'll see much more powerful tools for working with records later in this book.

Remember the type class `BoundedEnum` introduced in the previous chapter with the only method `range` which lists all the value constructors? It'll be convenient to use it in this project also; let's introduce a module with its definition:

Listing 3.4 BoundedEnum module

We thoroughly specify what we export from this module (type class and its method) using an *export list*.

```
module BoundedEnum (
    BoundedEnum (range)
  ) where

import Prelude (Enum (enumFrom), Bounded (minBound))

class (Enum a, Bounded a) => BoundedEnum a where
  range :: [a]
  range = enumFrom minBound
```

Module `Prelude` provides a huge number of names but we limit them to what we need.

Module `BoundedEnum` keeps importing and exporting to a bare minimum. Consequently, it's extremely resilient to any changes in the `base` package. We'll discuss a module's export and import sections in greater details in the next chapter.

3.3.2 Computing statistics

Computing statistics is the central part of this project. Two issues require serious consideration: which structure we expect the given data to have and what we provide as a result of computations. The header of the module `Statistics` states our intentions on its content:

```
{-# LANGUAGE DeriveAnyClass #-}
module Statistics (Statistic(..), StatEntry (..),
                   StatQFieldData, StatInfo, statInfo) where
```

We export only one function `statInfo` and several types which are required by the clients to report statistical information to the user.

The import section's rather small, but it can give you a clue on how we want the given data to be presented:

```
import Data.Ord (comparing)
import Data.Foldable (minimumBy, maximumBy)
import Data.Time (diffDays)

import BoundedEnum
import QuoteData
```

What do we need from the given data? We want to *map* over it and *fold* it to a single value, nothing else. Consequently, `(Functor t, Foldable t) ⇒ t QuoteData` should suffice. We could require `Traversable t` instead which conveniently extends both `Functor` and `Foldable` but there's no need to constraint our data beyond what's required.

Two dimensions are in the analysis: the chosen statistic (minimum, maximum, mean, and number of days between minimum and maximum) and the specific record field (`open`, `close`, `high`, `low`, and `volume`). Clearly, computing the minimum's the same for any field and extracting a field from the quote data doesn't depend on the computed statistic. We've already prepared the `QField` datatype that in mind and we should do the same for the statictics with the `Statistic` datatype.

The following listing presents types for the result of the computations: we define datatypes for the chosen statistics and one entry of statistic information, we also use type synonyms for tuples and lists to make up the whole statistic data:

```
data Statistic = Mean | Min | Max | Days
  deriving (Show, Eq, Enum, Bounded, BoundedEnum)

data StatEntry = StatEntry {
    stat :: Statistic,
    qfield :: QField,
    value :: Fixed4
  }

type StatQFieldData = (QField, [StatEntry])
type StatInfo = [StatQFieldData]
```

A tradeoff exists between defining datatypes and using type synonyms: it's often a matter of convenience. It's convenient to have accessor functions for the entries and use pattern matching on tuples and lists.

We're finally ready to do the computations. Minimums and maximums for any field can be computed by standard methods of the `Foldable` type class, but we should define functions for computing the mean values and the days spent between maximal and minimal values of a field.

Let's compute mean value first:

```
mean xs = sum xs / fromIntegral (length xs)
```

Note that this isn't an efficient implementation in the case of lists, as it demands to traverse the list twice. In general, a performance depends on the chosen `Fold-`

able instance: if the `length` implementation doesn't require traversing the whole data structure, then we're lucky enough to get an efficient implementation for free.

Computing the number of days can be done as follows:

Listing 3.5 Computing number of days

We use the function `Data.Time.diffDays` from the `time` package to compute the number of days between two dates.

```
daysBetween qf quotes = fromIntegral $ abs $ diffDays dMinQuote dMaxQuote
  where
    cmp = comparing (field2fun qf)
    dMinQuote = day $ minimumBy cmp quotes
    dMaxQuote = day $ maximumBy cmp quotes
```

We use the higher-order functions `minimumBy` and `maximumBy` from the `Data.Foldable` module.

The *comparator* `cmp` defined here by the function `Data.Ord.comparing` compares quotes by the given field.

> **NOTE** Note that I deliberately omit type signatures for these and the following auxiliary functions because types add nothing to understanding them. Remember that they aren't even exported from this module. This is like in programming languages with dynamic typing: we write what's important without bothering about unnecessary details. Nevertheless, if this code compiles it's type checked. We, the Haskellers, always thank static typing with type inference in Haskell.
>
> Can you infer types of these two functions by yourself? Think whether it's possible using information you have.

The rest of the solution in this subsection's an easy exercise in higher-order functions, mainly currying and partial application:

```
funcByField func qf = func . fmap (field2fun qf)

computeStatistic Mean = funcByField mean
computeStatistic Min = funcByField minimum
computeStatistic Max = funcByField maximum
computeStatistic Days = daysBetween
```

> **NOTE** We call a function *curried* whenever it takes only one argument and returns another function. The partiality of an application means that we give only part of the arguments leaving the rest unspecified. For example, function `funcByField` from the previous listing takes three arguments: a function, a field, and a collection of stock quotes data—but we use it as if it has only one.

The method `fmap` from the `Functor` type class is used in the definition of `funcByField` to extract the required slice of the quote data: we take the raw data collection and get a collection of values of a particular field, then we process it with the given folding function. This function depends on the statistic specified in the `computeStatistic` function.

Everything's ready for gathering the statistical information:

```
statInfo :: (Functor t, Foldable t) => t QuoteData -> StatInfo
statInfo quotes = map stQFData range
  where
    stQFData qf = (qf, [ StatEntry st qf v | st <- range,
                         let v = computeStatistic st qf quotes ])
```

In this function we go over the list of all statistics and generate a list of `StatQFieldData` with all the information we can compute.

Note one more point: every statistic value has type `Fixed4` although some values can well be integers (for example, the number of days or everything about volumes except the mean). We can think about a better (more general) type to represent statistics but here we meet another tradeoff: either being more expressive in types or keeping implementation simpler. I choose the latter but promise to fix the view of values that are expected to be integer when formatting report.

3.3.3 Formatting statistical report

The next goal's to format the report in text form. This is what we want to get as a result:

```
Statistics for Open:
  Mean: 138.0889
  Minimum: 106.5700
  Maximum: 164.8000
  Days between Min/Max: 290
Statistics for Close:
  Mean: 138.1829
  Minimum: 105.7100
  Maximum: 164.0500
  Days between Min/Max: 291
...
Statistics for Volume:
  Mean: 28199624.4664
  Minimum: 11475920
  Maximum: 111837300
  Days between Min/Max: 68
```

`StatInfo` already contains what we need and we only need to "pretty print" it. To do that we should explicitly say how to format:

- The name of every statistical characteristic
- The single statistical report entry
- The collection of all the `StatInfo` entries

The idea is to provide independent formatters and then combine them into the report.

The module `StatReport` starts as follows:

```
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE OverloadedStrings #-}
```

```
module StatReport (statReport, showStatEntryValue) where

import Data.Fixed (showFixed)
import Data.Text (Text)
import Fmt

import QuoteData
import Statistics
```

We use three GHC extensions that we already discussed, export the `statReport` function and an auxiliary `showStatEntryValue`, and import everything we need:

- `showFixed` from `Data.Fixed` to show values of `Fixed4` differently depending on whether we want to show trailing zeroes
- `Text` type from `Data.Text`—note that we don't need anything else apart from the name of the type
- the whole `Fmt` module from the `fmt` package—this is our formatting engine of choice
- this project's modules which we depend on

The simplest task is to format the names of the statistics, and we'll define a `Buildable` instance for the `Statistic` type:

```
instance Buildable Statistic where
  build Mean = "Mean"
  build Min = "Minimum"
  build Max = "Maximum"
  build Days = "Days between Min/Max"
```

Although we use `String` literals here, they're turned into `Builder` values (remember that `build` has type `p → Builder` for any `Buildable p`) thanks to the `OverloadedStrings` GHC extension and the fact that `Builder`implements the `IsString` type class we discussed in the previous chapter.

The logic for removing trailing zeroes is the following:

- number of days is always shown as integer
- minimal and maximal volumes are also shown as integers
- everything else requires decimal point with four digits precision (remember we use `Fixed4`)

This is how it's implemented, pattern matching record fields with `RecordWildCards` GHC extension makes it convenient:

```
showStatEntryValue :: StatEntry -> String
showStatEntryValue StatEntry {..} = showFixed (removeTrailing stat qfield)
     value
  where
    removeTrailing Days _ = True
    removeTrailing Min Volume = True
    removeTrailing Max Volume = True
    removeTrailing _ _ = False
```

The rest is a job for the `Fmt` module:

Listing 3.6 Formatting statistics report

We use prefix `""+|` and suffix `|+""` to make this expression more visually attractive, the operators `+|` and `|+` are responsible for calling the appropriate `build` function from `Buildable` for `Statistics` and `String`.

```
instance Buildable StatEntry where
  build se@StatEntry {..} = ""+|stat|+": "+|showStatEntryValue se|+""

instance Buildable StatQFieldData where
  build (qf, stats) = nameF ("Statistics for " +||qf||+"") $ unlinesF stats

statReport :: StatInfo -> Text
statReport = fmt . unlinesF
```

Function `nameF` creates heading for the several lines generated by `unlinesF` from the provided list of `StatEntry` values, `+||` and `||+` call `show` for the given `QField` between them.

We finish the report preparation by using `unlinesF` for the given list of `StatQFieldData` values, thismakes the calls to the right `build` functions.

The final call to `fmt` transforms `Builder` we've got out of the many `Buildable` values to `Text` and signals that we're done with the report. Note the following advantage: we have the flexibility to change individual formatters and the whole report view independently.

3.3.4 Drawing charts

I bet you expect a long subsection on drawing charts. Well, it's not. Fortunately, we have a good library at our disposal, which is able to do everything we want without saying too much.

Our goal here is to implement the following function to plot several fields from `QuoteData` in a chart and export it to a file:

```
plotChart :: (Functor t, Foldable t) =>
             String -> t QuoteData -> [QField] -> FilePath -> IO ()
```

The arguments are:

- The chart title
- The stock quote data collection
- The list of fields to be plotted
- The name of the generated file

The result is `IO ()` as we want to create a file inside this function.

The module `Charts` starts with a simple `module` declaration and a short list of imports:

```
module Charts (plotChart) where

import Data.Foldable (traverse_, toList)
import Graphics.Rendering.Chart.Easy (plot, line, (.=), layout_title)
```

```
import Graphics.Rendering.Chart.Backend.Diagrams (toFile,
                                                  loadSansSerifFonts,
                                                  FileOptions (..),
                                                  FileFormat (SVG))
import QuoteData
```

We'll have to perform three tasks here:

- turn our data into something which is expected by the `Chart` package
- set the required properties and content of the chart
- specify the file format we'd like to get

As for the first task, we could anticipate an inability to plot `Fixed4` values, but we can easily convert them to `Double` for plotting (with the help of `realToFrac`). In fact, it's possible to teach `Chart` to understand this new type (see the `PlotValue` type class for details) but converting what we have's a lot easier. We'll use the `Graphics.Rendering.Chart.Easy` module for getting the second task done. The *backend* `Graphics.Rendering.Chart.Backend.Diagrams` has everything we need for the third one. The following listing implements all of this:

Listing 3.7 Plotting chart

```
plotChart :: (Functor t, Foldable t) =>
             String -> t QuoteData -> [QField] -> FilePath -> IO ()
plotChart title quotes qfs fname = toFile fileOptions fname $ do
    layout_title .= title
    traverse_ plotLine qfs
  where
    fileOptions = FileOptions (800, 600) SVG loadSansSerifFonts
    plotLine qf = plot $ line (show qf)
                              [toList $ fmap (qf2pd qf) quotes]
    qf2pd qf q = (day q,
                  realToFrac $ field2fun qf q :: Double)
```

Provide a title for the chart with the `(.=)` operator.

Specify options for exporting a chart (size, format, and fonts to be used) to a file

Plot lines in the chart for every given field.

Plot a single line.

Convert the data collection to a list of values suitable for plotting (pairs of two axis values)

The `Day` value corresponds to the `x`-axis

Values of type `Fixed4` (taken from `QuoteData`) are converted to a `Double` (`y`-axis)

A few other points should be mentioned about this code. First, note the `do` block in the last argument of the `toFile` function. Looking at the type signature of `toFile` reveals that this `do` block works in the `EC` monad (skip the type classes in the type signature for now):

```
toFile :: (Default r, ToRenderable r) =>
          FileOptions -> FilePath -> EC r () -> IO ()
```

The good news is that you can write the code without bothering about this fact at all. Well, we have a `Monad`, which is enough.

The second point's the use of the `(.=)` operator which provides a title for the chart. This operator comes from the `lens` library, which is covered later in this book. `Chart` relies heavily on it, as do many modern Haskell packages,and it's important to grasp it once we are ready.

3.3.5 Designing user interface

As usual in Haskell, you should quickly turn the user input into something explicitly typed. In the case of command-line arguments (a list of `String` values) it's a good idea to parse them into a record. The package `optparse-applicative`, which we'll use in this section for parsing command-line arguments, follows exactly this approach. It's an example of a highly regarded, professional, purely declarative (thanks to good abstractions) package with great documentation and many use cases. I won't describe all of its features, but limit myself to a short demonstration.

Remember that we'd like to get the following user interface:

```
Usage: stockquotes FILE [-c|--company ARG] [-p|--prices] [-v|--volumes]
  Stock quotes data processing

Available options:
  FILE                     CSV file name
  -c,--company ARG         stock company's name
  -p,--prices              create file with prices chart
  -v,--volumes             create file with volumes chart
  -h,--help                Show this help text
```

Let's define the module `Params` for describing and parsing the command-line arguments. This module starts as follows:

```
module Params (Params (..), cmdLineParser) where

import Data.Semigroup ((<>))
import Options.Applicative
```

As for the `Params` datatype we'll need the name of a CSV file (mandatory argument), an optional company name for mentioning in charts and two switches which govern generating charts on share prices and volumes (we'd like to have separate charts for them due to incomparable scales):

```
data Params = Params {
                fname :: FilePath
              , company :: String
              , prices :: Bool
              , volumes :: Bool
              }
```

Note the layout style: you can often see it in Haskell (and other programming languages) as it makes easier adding or removing fields, we don't have to think about commas in the *previous* line.

For the most interesting part we should describe the correspondence between command-line arguments and this `Params` datatype. The package `optparse-applicative` uses `Applicative` and `Semigroup` type classes to do that, enabling a *declarative* description. We use this term *declarative* to express the fact that there are no processing steps in such a description. Instead we explicitly specify which `Params` field corresponds to (and, consequently, is constructed from) which argument. The following listing presents the definition of the `mkParams` function which does exactly that.

Listing 3.8 Describing correspondence between `Params` and command-line arguments

`Parser` is an `Applicative`, `Params` is a result of computations in this context.

We apply the multiparametric value constructor `Params` via the `<$>` operator from `Applicative`. There should be exactly four arguments (as in the `Params` record).

We use the `Semigroup` operation `<>` to combine all the properties of this command-line argument, namely its name (`metavar "FILE"`) and description (`help "…"`).

Function `strOption` defines an optional `String` argument.

We provide long and short argument names, description, and a default value by combining with `<>`.

Function `switch` defines an optional argument of type `Bool`.

Function `strArgument` from `Options.Applicative` at this position means that the first argument of `Params` is constructed from a mandatory argument of type `String` (here we use the fact that `FilePath` is a type synonym to `String`).

```
mkParams :: Parser Params
mkParams =
  Params <$>
      strArgument
      (metavar "FILE" <> help "CSV file name")
  <*> strOption
       (long "company" <> short 'c'
       <> help "stock company's name" <> value "")
  <*> switch
       (long "prices" <> short 'p'
        <> help "create file with prices chart")
  <*> switch
       (long "volumes" <> short 'v'
        <> help "create file with volumes chart")
```

Note the pattern: all the functions `strArgument`, `strOption`, `switch` take `Semigroup`-based combinations of properties and every such function refers to exactly one `Params` field at the same position.

Now that we've described the correspondence between command-line arguments and `Params` fields we should augment it with additional usage information which is printed if the user specifies `--help` or `-h` switches and run the parsing. The following listing demonstrates how to do that.

Listing 3.9 Command-line arguments parser

This is an `IO` action because we need access to command-line arguments, the result of the computation has type `Params`.

Function `execParser` from the `Options.Applicative` module runs parsing.

```
cmdLineParser :: IO Params
cmdLineParser = execParser opts
  where
    opts = info (mkParams <**> helper)
                (fullDesc <> progDesc "Stock quotes data processing")
```

We augment `mkParams` with the switches for displaying help information.

We specify a short program description with `progDesc` and ask to generate a full description by mentioning `fullDesc` in this combination of properties.

3.3.6 Connecting the parts

The last thing to do is to connect all the parts of this example, namely we should:

- get `Params` from command-line arguments via `cmdLineParser`
- read the CSV file
- compute the statistics
- prepare and print the report
- generate the charts if required

We'll do all that in the `Main` module which starts as follows:

```
{-# LANGUAGE RecordWildCards #-}

module Main where

import Control.Monad (when)
import qualified Data.Text.IO as TIO
import qualified Data.ByteString.Lazy as BL (readFile)
import Data.Csv (decodeByName)

import QuoteData
import Statistics
import StatReport
import Charts
import Params
```

We'll split the job between three functions: `main`, `work`, and `generateReports`. The `main` function's responsible for running the command-line parser and delegates everything else to `work`:

```
main :: IO ()
main = cmdLineParser >>= work
```

The `work` function takes the constructed `Params` as an argument, reads and decodes the CSV file, and runs `generateReports` whenever everything goes well:

```
work :: Params -> IO ()
work params = do
  csvData <- BL.readFile (fname params)
  case decodeByName csvData of
    Left err -> putStrLn err
    Right (_, quotes) -> generateReports params quotes
```

We read a `ByteString` (from `Data.ByteString.Lazy`, imported with the prefix `BL`) from the file and decode it with the `decodeByName` function from the `cassava` package's `Data.Csv` module. This function has the following type signature:

```
decodeByName :: FromNamedRecord a
             => BL.ByteString
             -> Either String (Header, Vector a)
```

Because `quotes` is later used as a value of type `(Functor t, Foldable t) ⇒ t QuoteData`, the type checker can figure out that the `a` type variable in the type signature for `decodeByName` refers to `QuoteData`, which already has an instance of `FromNamedRecord` thanks to the `deriving` clause in its definition.

Type `Vector` comes from the `vector` package, which provides an efficient implementation of `Int`-indexed arrays with many optimisations for loop-like operations. We'll see vectors at work in other examples later in this book.

You see that in the case of correct decoding we get a vector of `QuoteData` values. `Vector` implements both `Functor` and `Foldable` type classes, and all our code for computing statistics and building reports remains intact (but extremely efficient thanks to `Vector` instances of `Functor` and `Foldable`).

The following listing presents the `generateReports` function which does the rest of the job.

Listing 3.10 Generating reports

GHC extension `RecordWildCards` makes the `Params` record fields accessible through out the code.

```
generateReports :: (Functor t, Foldable t) => Params -> t QuoteData -> IO ()
generateReports Params {..} quotes = do
  TIO.putStr $ statReport statInfo'
  when prices $ plotChart title quotes [Open, Close, High, Low] fname_prices
  when volumes $ plotChart title quotes [Volume] fname_volumes
 where
   statInfo' = statInfo quotes
   withCompany pref  = if company /= "" then pref ++ company else ""
   img_suffix = withCompany "_" ++ ".svg"
   fname_prices = "prices" ++ img_suffix
   fname_volumes = "volumes" ++ img_suffix
   title = "Historical Quotes" ++ withCompany " for "
```

Variable `company` comes from the `Params` record field.

We encode here the following logic for chart file names:

- they are `prices.svg` and `volumes.svg` by default
- if the user specifies a company name, say `"AAPL"`, then these names will be `prices_AAPL.svg` and `volumes_AAPL.svg` respectively

NOTE Clearly this decision to hard code file names isn't the best one. We do that only with simplicity in mind. If we give the user option to specify filenames then we should think about filename correctness, file extensions, various image file formats, etc. You're justified to do it right.

This concludes the implementation. We've seen many rather basic Haskell features, that you meet all the time. In the rest of this chapter we'll see how convenient it is to extend this project with new functionality.

3.4 Extending project with reports in HTML

Let's set a new task: we want to generate a report in HTML format. This new report should include:

- chart images (if requested)
- a statistical report (with the same information as before)
- a table with all the stock quotes data we've read from the CSV file

We should also allow the user to switch off the text report as this information's presented in HTML.

To accomplish this task we should do some changes in `Params` and in the `Main` modules, and add new module `HtmlReport`. Remember in the previous section we also added the `BoundedEnum` module. Figure 3.7 presents the final module structure for this project.

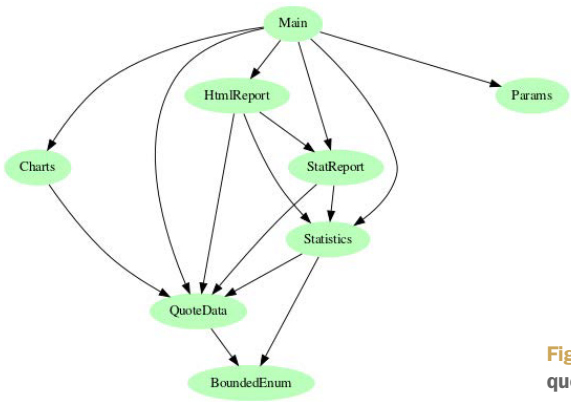


Figure 3.7 Module structure for stock quote data example: final version

We'll extend the project iteratively without breaking anything.

3.4.1 Changes in user interface

The user interface's described in the module `Params`. We should add a switch for generating the report in HTML (with a corresponding field in `Params` record) and a switch to disable the text report (with a field as well). The new `Params` record looks as follows:

Listing 3.11 Record `Params` extended for HTML report generation

```
data Params = Params {
                fname :: FilePath
              , company :: String
              , prices :: Bool
              , volumes :: Bool
              , html :: Bool
              , no_text :: Bool
              }
```

The changes to `mkParams` are also straightforward, we should append the following four lines to its definition:

```
      <*> switch
          (long "html" <> help "create file with HTML report")
      <*> switch
          (long "no-text" <> short 'n' <> help "don't print statistics
    report")
```

Thanks to the declarative nature of describing command-line parsers, everything else in this module remains intact. Running this program with flag `-h` gives the following output:

```
Usage: stockquotes FILE [-c|--company ARG] [-p|--prices] [-v|--volumes] [--
    html]
                   [-n|--no-text]
  Stock quotes data processing

Available options:
  FILE                     CSV file name
  -c,--company ARG         stock company's name
  -p,--prices              create file with prices chart
  -v,--volumes             create file with volumes chart
  --html                   create file with HTML report
  -n,--no-text             don't print statistics report
  -h,--help                Show this help text
```

Note that we haven't changed anything else apart from the `Params` module yet.

3.4.2 Generating reports in HTML format with blaze-html

I won't give you a thorough description of the package `blaze-html` that we'll use for generation report in HTML. Instead I'll comment on the solution in this project. Refer to the documentation on Hackage (https://hackage.haskell.org/package/blaze-html) if you need more details.

We'll need the following to generate report in HTML:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}

module HtmlReport (htmlReport) where

import Data.Foldable (traverse_)
import Data.Semigroup ((<>))
import Data.ByteString.Lazy (ByteString)
import Text.Blaze.Html5 as H
import Text.Blaze.Html5.Attributes (src)
import Text.Blaze.Html.Renderer.Utf8 (renderHtml)

import QuoteData
import StatReport (showStatEntryValue)
import Statistics
import Fmt
```

The `blaze-html` package relies heavily on the `OverloadedStrings` GHC extension to turn every `String` literal into the `Html` type. Module `Text.Blaze.Html5` provides the convenience functions `string` and `text` for converting `String` and `Text` values into values of `Html` type. Technically, these functions comes from `Text-Blaze` module from another package `blaze-markup` and are reexported by `Text.Blaze.Html5`.

Note the line `import Text.Blaze.Html5 as H`: we use such imports for optional qualifying names that comes from the module. The qualification solves the problem of name clashing but we're free to omit it whenever there's no chance for clash.

Our goal's to implement the function `htmlReport`:

```
htmlReport :: (Functor t, Foldable t) =>
              String -> t QuoteData -> StatInfo -> [FilePath] -> ByteString
```

with the following arguments:

- HTML document title
- `QuoteData` collection
- Statistical information
- List of files with generated chart images

The HTML report's generated as `ByteString` to be saved in a file later. Note, that we're able to do the report generation purely. The plan is the following:

1. Describe the HTML document with `head` and `body`.
2. Put the document title and style descriptions into `head`.
3. Put all the generated images with charts into `img` tags.
4. Add a `table` with the statistical information.
5. Add a `table` with the stock quote data.

Look how cleanly this plan maps to the code:

```
htmlReport title quotes si images = renderHtml $ docTypeHtml $ do
     H.head $ do
       H.title $ string title
       H.style style
     body $ do
       renderDiagrams images
       renderStatInfo si
       renderData quotes
  where
    ...
```

Note the `do` blocks in this code: all the functions `docTypeHtml`, `head`, `title`, `style`, and `body` have the same type `Html → Html` where the argument's referred as *inner HTML*: they provide a wrapper! `Html` is in fact a `Monad` and we can use `do` block (and all the other monadic machinery) to work with it.

Every other function and value resides in `where` block, for example the following style definition for presenting HTML tables:

```
style = "table {border-collapse: collapse}" <>
        "td, th {border: 1px solid black; padding: 3px}"
```

The `Semigroup` operation `<>` is used here for combining `Html` values (well, `Html` isn't only a `Monad` it's also a `Semigroup`, check and mate, dear Front End Developers).

Rendering section with diagrams is straightforward:

```
renderDiagrams [] = pure ()
renderDiagrams images = do
  h1 "Diagrams"
  traverse_ ((img!).src.toValue) images
```

We omit this section completely if no image file name's present. If we have any images then we output section the title and turn every image file name into an `img` HTML tag with the `src` attribute set to the file name. A call to `renderDiagrams ["prices.svg", "volumes.svg"]` would produce:

```
<h1>Diagrams</h1><img src="prices.svg"/><img src="volumes.svg"/>
```

Generating statistical information in HTML is a bit more involved:

```
renderStatInfo [] = pure ()
renderStatInfo si@((_, ses):_) = do
  h1 "Statistics Report"
  table $ do
     thead $ tr $ traverse_ th
           $ "Quotes Field" : [text $ fmt $ build $ stat s | s <- ses]
     tbody $ traverse_ statData2TR si

statData2TR (qf, entries) = tr $ do
  td $ string $ show qf
  traverse_ (td.string.showStatEntryValue) entries
```

We use here the `Fmt` and `StatReport` modules to avoid repeating the formatting rules that we've defined previously.

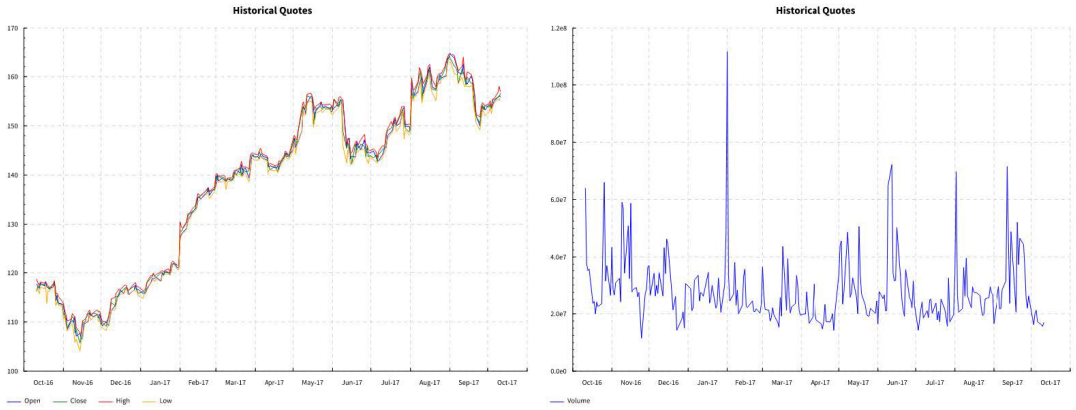
Finally, we need to render the table with all the stock quotes data:

```
renderData quotes = do
  h1 "Stock Quotes Data"
  table $ do
     thead $ tr
           $ traverse_ th ["Day", "Close", "Volume", "Open", "High", "Low"]
     tbody $ traverse_ quoteData2TR quotes

quoteData2TR QuoteData {..} = tr $ do
  td $ string $ show day
  traverse_ (td.string.show) [close, volume, open, high, low]
```

Figure 3.8 demonstrates the exact view of the report we can get as a result.

Diagrams



Statistics Report

Quotes Field	Mean	Minimum	Maximum	Days between Min/Max
Open	138.0889	106.5700	164.8000	290
Close	138.1829	105.7100	164.0500	291
High	138.9951	107.6800	164.9400	290
Low	137.1893	104.0800	163.6300	291
Volume	28199624.4664	11475920	111837300	68

Stock Quotes Data

Day	Close	Volume	Open	High	Low
2017-10-11	156.5500	16861450.0000	155.9700	156.9800	155.7500
2017-10-10	155.9000	15603520.0000	156.0550	158.0000	155.1000
2017-10-09	155.8400	16243080.0000	155.8100	156.7300	155.4850
2017-10-06	155.3000	17223790.0000	154.9700	155.4900	154.5600
2017-10-05	155.3900	21215870.0000	154.1800	155.4400	154.0500
2017-10-04	153.4800	20088940.0000	153.6300	153.8600	152.4600
2017-10-03	154.4800	16216800.0000	154.0100	155.0900	153.9100
2017-10-02	153.8100	18631540.0000	154.2600	154.4500	152.7200

Figure 3.8 Html report for stockquotes project

3.4.3 Changes in the `Main` module

To conclude this example, we should make several changes to `Main`. First, let's check whether to print text report in the `do` block of the `generateReports` function:

```
unless no_text $ TIO.putStr $ statReport statInfo'
```

The `unless` function comes from `Control.Monad`, and we should tweak the corresponding `import` accordingly.

Second, we should prepare the list of image file names and the file name for the HTML report. The former should take into account which charts are generated; let's introduce the following lines into the `where`block of `generateReports`:

```
images = concat $ zipWith (bool []) [[fname_prices],[fname_volumes]]
                                    [prices, volumes]
fname_html = "report" ++ withCompany "_" ++ ".html"
```

The `bool` function comes from `Data.Bool` (we should import it), here is its type:

```
bool :: a -> a -> Bool -> a
```

It returns the first argument if `Bool` is `False` and the second one otherwise. We're using it to remove the file name of the chart that wasn't requested by the user.

Finally, we should import the `HtmlReport` module, and generate and save the HTML report into a file (in case it was requested by the user):

```
when html $ BL.writeFile fname_html $ htmlReport title quotes statInfo'
   images
```

This is everything you need. You see that there were rather limited changes. It'd be useful to extend this project somehow on your own. Why not generate some bar charts or do more sophisticated analysis?

3.5 Summary

- Use the package `time` whenever processing dates and times.
- Choose your own favorite package for representing textual data: `formatting` and `fmt` are good candidates.
- Drawing charts is easy with the `Chart` package; give it a try!
- Try the `cassava` package for parsing CSV files.
- Use the `optparse-applicative` package for parsing command-line arguments and creating default help screens.
- Learn to build HTML documents with the `blaze-html` package.
- Explore and use functions from the `Safe` module, they're better than that which is provided by `Prelude`.

index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

Z


manning