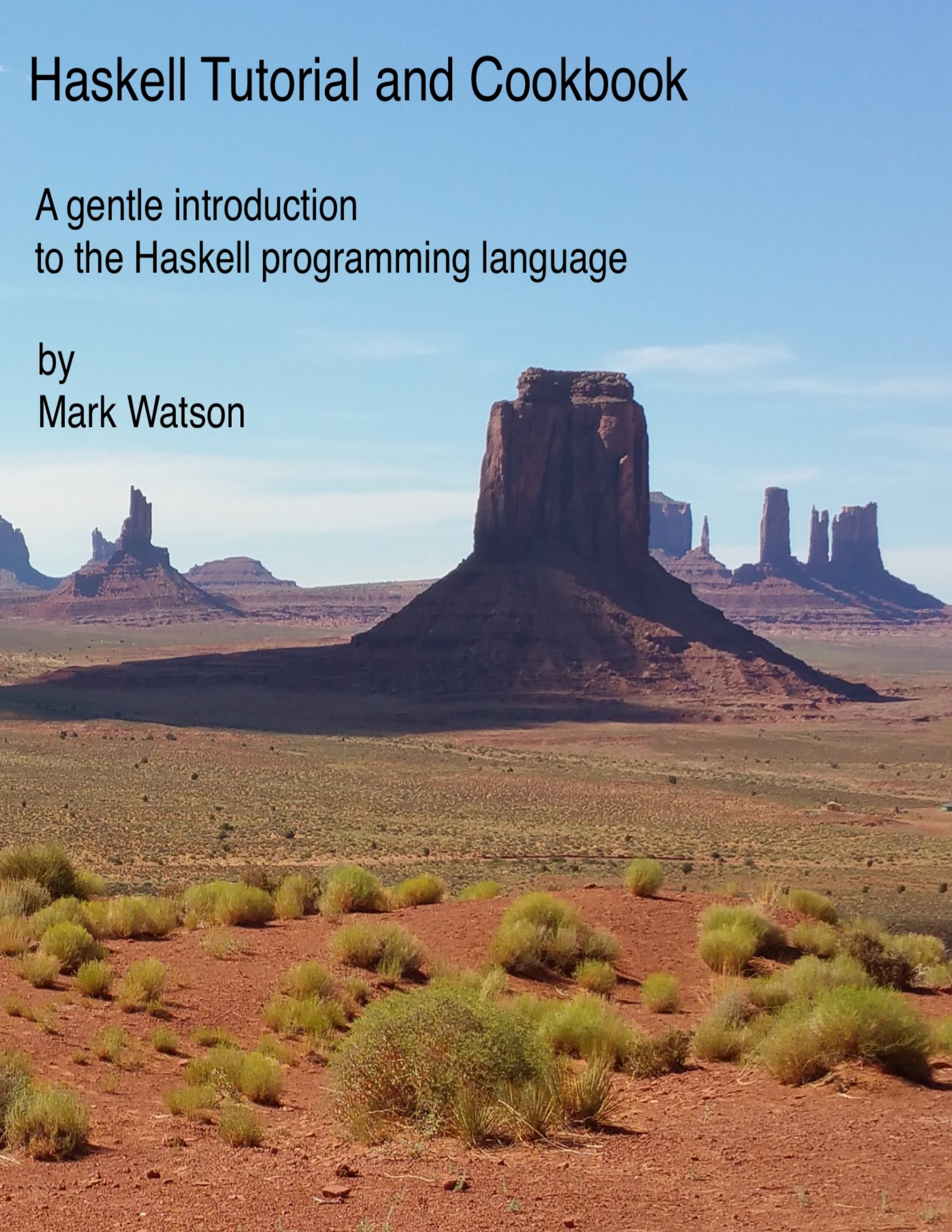
Haskell Tutorial and Cookbook

A gentle introduction
to the Haskell programming language

by
Mark Watson



Haskell Tutorial and Cookbook

Mark Watson

This book is for sale at http://leanpub.com/haskell-cookbook

This version was published on 2016-11-18

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Cover Material, Copyright, and License

This eBook will be updated occasionally so please periodically check the leanpub.com web page for this book[1] for updates.

Please visit the author’s website[2].

If you found a copy of this book on the web and find it of value then please consider buying a copy at leanpub.com/haskell-cookbook[3] to support the author and fund work for future updates.

[1] https://leanpub.com/haskell-cookbook
[2] http://markwatson.com
[3] https://leanpub.com/haskell-cookbook

Preface

It took me over a year learning Haskell before I became comfortable with the language because I tried to learn too much at once. There are two aspects to Haskell development: writing pure functional code and writing impure code that needs to maintain state and generally deal with the world non-deterministically. I usually find writing pure functional Haskell code to be easy and a lot of fun. Writing impure code is sometimes a different story. This is why I am taking a different approach to teaching you to program in Haskell: we begin techniques for writing concise, easy to read and understand efficient pure Haskell code. I will then show you patterns for writing impure code to deal with file IO, network IO, database access, and web access. You will see that the impure code tends to be (hopefully!) a small part of your application and is isolated in the impure main program and in a few impure helper functions used by the main program. Finally, we will look at a few larger Haskell programs.

A Request from the Author

I spent time writing this book to help you, dear reader. I release this book under the Creative Commons "share and share alike, no modifications, no commercial reuse" license and set the minimum purchase price to $4.00 in order to reach the most readers. Under this license you can share a PDF version of this book with your friends and coworkers. If you found this book on the web (or it was given to you) and if it provides value to you then please consider doing one of the following to support my future writing efforts and also to support future updates to this book:

- Purchase a copy of this book at leanpub.com/haskell-cookbook[4]
- Hire me as a consultant[5]

I enjoy writing and your support helps me write new editions and updates for my books and to develop new book projects. Thank you!

I am currently working on a second volume of this book, with less tutorial material and more code recipes for web development, command line utilities, more artificial intelligence code, etc. Here is the web page for "Haskell Cookbook, Volume 2"[6].

[4] https://leanpub.com/haskell-cookbook
[5] http://markwatson.com/
[6] https://leanpub.com/haskellcookbook2

Structure of the Book

The first section of this book contains two chapters:

- A tutorial on pure Haskell development: no side effects.
- A tutorial on impure Haskell development: dealing with the world (I/O, network access, database access, etc.). This includes examples of file IO and network programming as well as writing short applications: a mixture of pure and impure Haskell code.

After working through these tutorial chapters you will understand enough of Haskell development to understand and be able to make modifications for your own use of the cookbook examples in the second section. Some of the general topics will be covered again in the second book section that contains longer sample applications. For example, you will learn the basics for interacting with Sqlite and Postgres databases in the tutorial on impure Haskell code but you will see a much longer example later in the book when I provide code that implements a natural language processing (NLP) interface to relational databases.

The second section of this book contains the following recipes implemented as complete programs:

- Textprocessing CSV Files
- Textprocessing JSON Files
- Natural Language Processing (NLP) interface to relational databases, including annotating English text with Wikipedia/DBPedia URIs for entities in the original text. Entities can be people, places, organizations, etc.
- Accessing and Using Linked Data
- Querying Semantic Web RDF Data Sources
- Web scraping data on web sites
- Using Sqlite and Postgres relational databases
- Play a simple version of Blackjack card game

Code Examples

The code examples in this book are licensed under two software licenses and you can choose the license that works best for your needs: Apache 2 and GPL 3. To be clear, you can use the examples in commercial projects under the Apache 2 license and if you like to write Free (Libre) software then use the GPL 3 license.

We will use *stack* as a build system for all code examples. The code examples are provided as 15 separate *stack* based projects. These examples are found on github[7].

[7] https://github.com/mark-watson/haskell_tutorial_cookbook_examples

Functional Programming Requires a Different Mind Set

You will learn to look at problems differently when you write functional programs. We will use a bottom up approach in most of the examples in this book. I like to start by thinking of the problem domain and decide how I can represent the data required for the problem at hand. I prefer to use native data structures. This is the opposite approach to object oriented development where considerable analysis effort and coding effort is required to define class hierachies to represent data. In most of the code we use simple native data types like lists and maps.

Once we decide how to represent data for a program we then start designing and implementing simple functions to operate on and transform data. If we find ourselves writing functions that are too long or too complex, we can break up code into simpler functions. Haskell has good language support for composing simple functions into more complex operations.

I have spent many years engaged in object oriented programming starting with CLOS for Common Lisp, C++, Java, and Ruby. I now believe that in general, and I know it is sometimes a bad idea to generalize too much, functional programming is a superior paradigm to object oriented programming. Convincing you of this belief is one of my goals in writing this book!

eBooks Are Living Documents

I wrote printed books for publishers like Springer-Verlag, McGraw-Hill, and Morgan Kaufman before I started self-publishing my own books. I prefer eBooks because I can update already published books and update the code examples for eBooks.

I encourage you to periodically check for updates to both this book and the code examples on the leanpub.com web page for this book[8].

Setting Up Your Development Environment

I strongly recommend that you use the *stack* tool from the stack website[9]. This web site has instructions for installing *stack* on OS X, Windows, and Linux. If you don't have *stack* installed yet please do so now and follow the "getting started" instructions for creating a small project. Appendix A contains material to help get you set up.

It is important for you to learn the basics of using *stack* before jumping into this book because I have set up all of the example programs using stack.

The github repository for the examples in this book is [located hereare found on github[10].

[8] https://leanpub.com/haskell-cookbook

[9] http://docs.haskellstack.org/en/stable/README.html

[10] https://github.com/mark-watson/haskell_tutorial_cookbook_examples

Many of the example listings for code examples are partial or full listing of files in my github repository. I show the file name, the listing, and the output. To experiment with the example yourself you need to load it and execute the main function; for example, if the example file is TestSqLite1.hs in the sub-directory Database, then from the top level directory in the git repository for the book examples you would do the following:

```
$ haskell_tutorial_cookbook_examples git:(master) > cd Database
$ Database git:(master) > stack build --exec ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :l TestSqLite1
[1 of 1] Compiling Main             ( TestSqLite1.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
"Table names in database test.db:"
"test"
"SQL to create table 'test' in database test.db:"
"CREATE TABLE test (id integer primary key, str text)"
"number of rows in table 'test':"
1
"rows in table 'test':"
(1,"test string 2")
*Main>
```

If you don't want to run the example in a REPL in order to experiment with it interactively then you can just run it via stack using:

```
$ Database git:(master) > stack build --exec TestSqlite1
"Table names in database test.db:"
"test"
"SQL to create table 'test' in database test.db:"
"CREATE TABLE test (id integer primary key, str text)"
"number of rows in table 'test':"
1
"rows in table 'test':"
(1,"test string 2")
```

I include *README.md* files in the project directories with specific instructions.

If you are an Emacs user I recommend that you follow the instructions in Appendix A, load the tutorial files into an Emacs buffer, build an example and open a REPL frame. If one is not already open type control-c control-l, switch to the REPL frame, and run the **main** function. When you make changes to the tutorial files, doing another control-c control-l will re-build the example in less than

a second. In addition to using Emacs I occasionally use the IntelliJ Community Edition (free) IDE with the Haskell plugin, the TextMate editor (OS X only) with the Haskell plugin, or the GNU GEdit editor (Linux only).

Whether you use Emacs or run a REPL in a terminal window (command window if you are using Windows) the important thing is to get used to and enjoy the interactive style of development that Haskell provides.

Why Haskell?

I have been using Lisp programming languages professionally since 1982. Lisp languages are flexible and appropriate for many problems. Some might dissagree with me but I find that Haskell has most of the advantages of Lisp with the added benefit of being strongly typed. Both Lisp and Haskell support a style of development using an interactive shell (or "repl").

What does being a strongly typed language mean? In a practical sense it means that you will often encounter syntax errors caused by type mismatches that you will need to fix before your code will compile (or run in the GHCi shell interpreter). Once your code compiles it will likely work, barring a logic error. The other benefit that you can get is having to write fewer unit tests - at least that is my experience. So, using a strongly typed language is a tradeoff. When I don't use Haskell I tend to use dynamic languages like Clojure and Ruby.

Enjoy Yourself

I have worked hard to make learning Haskell as easy as possible for you. If you are new to the Haskell programming language then I have something to ask of you, dear reader: please don't rush through this book, rather take it slow and take time to experiment with the programming examples that most interest you.

Section 1 - Tutorial

The first section of this book contains two chapters:

- A tutorial on pure Haskell development: no side effects.
- A tutorial on impure Haskell development: dealing with the world (I/O, network access, database access, etc.)

After working through these two tutorial chapters you will have sufficient knowledge of Haskell development to understand the cookbook examples in the second section and be able to modify them for your own use. Some of the general topics will be covered again in the second book section that contains longer example programs.

Tutorial on Pure Haskell Programming

Pure Haskell code has no side effects and if written properly is easy to read and understand. I am assuming that you have installed *stack* using the directions in Appendix A. It is important to keep a Haskell interactive **repl** open as you read the material in this book and experiment with the code examples as you read. I don't believe that you will be able to learn the material in this chapter unless you work along trying the examples and experimenting with them in an open Haskell repl!

The directory **Pure** in the git repository contains the examples for this chapter. Many of the examples contain a small bit of impure code in a **main** function. We will cover how this impure code works in the next chapter. Here is an example of impure code, contained insde a **main** function that you will see in this chapter:

```
main = do
  putStrLn ("1 + 2 = " ++ show (+ 1 2))
```

I ask you to treat these small bits of impure code in this chapter as a "black box" and wait for the next chapter for a fuller explanation.

Pure Haskell code performs no I/O, network access, access to shared in-memory datastructures, etc.

The first time you build an example program with *stack* it may take a while since library dependencies need to be loaded from the web. In each example directory, after an initial **stack build** or **stack ghci** (to run the repl) then you should not notice this delay.

Interactive GHCi Shell

The interactive shell (often called a "repl") is very useful for learning Haskell: understanding types and the value of expressions. While simple expressions can be typed directly into the GHCi shell, it is usually better to use an external text editor and load Haskell source files into the shell (repl). Let's get started. Assuming that you have installed *stack* as described in Appendix A, please try:

```
1  ~/$ cd haskell_tutorial_cookbook_examples/Pure
2  ~/haskell_tutorial_cookbook_examples/Pure$ stack ghci
3  Using main module: Package `Pure' component exe:Simple with main-is file: /home/\
4  markw/BITBUCKET/haskell_tutorial_cookbook_examples/Pure/Simple.hs
5  Configuring GHCi with the following packages: Pure
6  GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
7  [1 of 1] Compiling Main             ( /home/markw/BITBUCKET/haskell_tutorial_coo\
8  kbook_examples/Pure/Simple.hs, interpreted )
9  Ok, modules loaded: Main.
10 *Main> 1 + 2
11 3
12 *Main> (1 + 2)
13 3
14 *Main> :t (1 + 2)
15 (1 + 2) :: Num a => a
16 *Main> :l Simple.hs
17 [1 of 1] Compiling Main             ( Simple.hs, interpreted )
18 Ok, modules loaded: Main.
19 *Main> main
20 1 + 2 = 3
21 *Main>
```

If you are working in a repl and edit a file you just loaded with **:l**, you can then reload the last file loaded using **:r** without specifying the file name. This makes it quick and easy to edit a Haskell file with an external editor like Emacs or Vi and reload it in the repl after saving changes to the current file.

Here we have evaluated a simple expression "1 + 2" in line 10. Notice that in line 12 we can always place parenthesis around an expression without changing its value. We will use parenthesis when we need to change the default orders of precedence of functions and operators and make the code more readable.

In line 14 we are using the ghci **:t** command to show the type of the expression **(1 + 2)**. The type **Num** is a type class (i.e., a more general purpose type that other types can inherit from) that contains several sub-types of numbers. As examples, two subtypes of **Num** are **Fractional** (e.g., 3.5) and **Integer** (e.g., 123). Type classes provide a form of function overloading since existing functions can be redefined to handle arguments that are instances of new classes.

In line 16 we are using the ghci command **:l** to load the external file *Simple.hs*. This file contains a function called **main** so we can execute **main** after loading the file. The contents of *Simple.hs* is:

```
1  module Main where
2
3  sum2 x y = x + y
4
5  main = do
6    putStrLn ("1 + 2 = " ++ show (sum2 1 2))
```

Line 1 defines a module named **Main**. The rest of this file is the definition of the module. This form of the module **do** expression exports all symbols so other code loading this module has access to **sum2** and **main**. If we only wanted to export **main** then we could use:

```
module Main (main) where
```

The function **sum2** takes two arguments and adds them together. I didn't define the type of this function so Haskell does it for us using type inference.

```
1  *Main> :l Simple.hs
2  [1 of 1] Compiling Main             ( Simple.hs, interpreted )
3  Ok, modules loaded: Main.
4  *Main> :t sum2
5  sum2 :: Num a => a -> a -> a
6  *Main> sum2 1 2
7  3
8  *Main> sum2 1.0 2
9  3.0
10 *Main> :t 3.0
11 3.0 :: Fractional a => a
12 *Main> :t 3
13 3 :: Num a => a
14 *Main> (toInteger 3)
15 3
16 *Main> :t (toInteger 3)
17 (toInteger 3) :: Integer
18 *Main>
```

What if you want to build a standalone executable program from the example in **Smple.hs**? Here is an example:

```
$ stack ghc Simple.hs
[1 of 1] Compiling Main             ( Simple.hs, Simple.o )
Linking Simple ...
$ ./Simple
1 + 2 = 3
```

Most of the time we will use simple types built into Haskell: **characters**, **strings**, **lists**, and **tuples**. The type **Char** is a single character. One type of string is a list of characters **[Char]**. (Another type **ByteString** will be covered in later chapters.) Every element in a list must have the same type. A **Tuple** is like a list but elements can be different types. Here is a quick introduction to these types, with many more examples later:

```
*Main> :t 's'
's' :: Char
*Main> :t "tree"
"tree" :: [Char]
*Main> 's' : "tree"
"stree"
*Main> :t "tick"
"tick" :: [Char]
*Main> 's' : "tick"
"stick"
*Main> :t [1,2,3,4]
[1,2,3,4] :: Num t => [t]
*Main> :t [1,2,3.3,4]
[1,2,3.3,4] :: Fractional t => [t]
*Main> :t ["the", "cat", "slept"]
["the", "cat", "slept"] :: [[Char]]
*Main> ["the", "cat", "slept"] !! 0
"the"
*Main> head ["the", "cat", "slept"]
"the"
*Main> tail ["the", "cat", "slept"]
["cat","slept"]
*Main> ["the", "cat", "slept"] !! 1
"cat"
*Main> :t (20, 'c')
(20, 'c') :: Num t => (t, Char)
*Main> :t (30, "dog")
(30, "dog") :: Num t => (t, [Char])
*Main> :t (1, "10 Jackson Street", 80211, 77.5)
(1, "10 Jackson Street", 80211, 77.5)
  :: (Fractional t2, Num t, Num t1) => (t, [Char], t1, t2)
```

The GHCi repl command **:t** tells us the type of any expression or function. Much of your time developing Haskell will be spent with an open repl and you will find yourself checking types many times during a development session.

In line 1 you see that the type of **'s'** is **'s'** :: **Char** and in line 3 that the type of the string "tree" is **[Char]** which is a list of characters. The abbreviation **String** is defined for **[Char]**; you can use either. In line 9 we see the "cons" operator : used to prepend a character to a list of characters. The cons : operator works with all types contained in any lists. All elements in a list must be of the same type.

The type of the list of numbers **[1,2,3,4]** in line 11 is **[1,2,3,4]** :: **Num t** ⇒ **[t]**. The type **Num** is a general number type. The expression **Num t** ⇒ **[t]** is read as: "*t* is a type variable equal to **Num** and the type of the list is **[t]**, or a list of Num values". It bears repeating: all elements in a list must be of the same type. The functions **head** and **tail** used in lines 19 and 21 return the first element of a list and return a list without the first element.

You will use lists frequently but the restriction of all list elements being the same type can be too restrictive so Haskell also provides a type of sequence called **tuple** whose elements can be of different types as in the examples in lines 25-31.

Tuples of length 2 are special because functions **fst** and **snd** are provided to access the first and second pair value:

```
*Main> fst (1, "10 Jackson Street")
1
*Main> snd (1, "10 Jackson Street")
"10 Jackson Street"
*Main> :info fst
fst :: (a, b) -> a         -- Defined in ‘Data.Tuple’
*Main> :info snd
snd :: (a, b) -> b         -- Defined in ‘Data.Tuple’
```

Please note that **fst** and **snd** will not work with tuples that are not of length 2. Also note that if you use the function **length** on a tuple, the result is always one because of the way tuples are defined as Foldable types, which we will use later.

Haskell provides a concise notation to get values out of long tuples. This notation is called destructuring:

```
1  *Main> let geoData = (1, "10 Jackson Street", 80211, 77.5)
2  *Main> let (_,_,zipCode,temperature) = geoData
3  *Main> zipCode
4  80211
5  *Main> temperature
6  77.5
```

Here, we defined a tuple **geoData** with values: index, street address, zip code, and temperature. In line two we extract the zip code and temperature. Another reminder: we use **let** in lines 1-2 because we are in a repl.

Like all programming languages, Haskell has operator precedence rules as these examples show:

```
1  *Main> 1 + 2 * 10
2  21
3  *Main> 1 + (2 * 10)
4  21
5  *Main> length "the"
6  3
7  *Main> length "the" + 10
8  13
9  *Main> (length "the") + 10
10 13
```

The examples in lines 1-4 illustrate that the multiplication operator has a higher precidence than the addition operator.

```
*Main> :t length
length :: Foldable t => t a -> Int
*Main> :t (+)
(+) :: Num a => a -> a -> a
```

Note that the function **length** starts with a lower case letter. All Haskell functions start with a lower case letter except for type constructor functions that we will get to later. A **Foldable** type can be iterated through and be processed with map functions (which we will use shortly).

We saw that the function **+** acts as an infix operator. We can convert infix functions to prefix functions by enclosing them in parenthesis:

```
*Main> (+) 1 2
3
*Main> div 10 3
3
*Main> 10 `div` 3
3
```

In this last example we also saw how a prefix function **div** can be used infix by enclosing it in back tick characters.

```
1  *Main> let x3 = [1,2,3]
2  *Main> x3
3  [1,2,3]
4  *Main> let x4 = 0 : x3
5  *Main> x4
6  [0,1,2,3]
7  *Main> x3 ++ x4
8  [1,2,3,0,1,2,3]
9  *Main> x4
10 [0,1,2,3]
11 *Main> x4 !! 0
12 0
13 *Main> x4 !! 100
14 *** Exception: Prelude.!!: index too large
15 *Main> let myfunc1 x y = x ++ y
16 *Main> :t myfunc1
17 myfunc1 :: [a] -> [a] -> [a]
18 *Main> myfunc1 x3 x4
19 [1,2,3,0,1,2,3]
```

Usually we define functions in files and load them as we need them. Here is the contents of the file myfunc1.hs:

```
1  myfunc1 :: [a] -> [a] -> [a]
2  myfunc1 x y = x ++ y
```

The first line is a type signature for the function and is not required; here the input arguments are two lists and the output is the two lists concatenated together. In line 1 note that **a** is a type variable that can represent any type. However, all elements in the two function input lists and the output list are constrained to be the same type.

```
*Main> :l myfunc1.hs
[1 of 1] Compiling Main             ( myfunc1.hs, interpreted )
Ok, modules loaded: Main.
*Main> myfunc1 ["the", "cat"] ["ran", "up", "a", "tree"]
["the","cat","ran","up","a","tree"]
```

Please note that the *stack* repl auto-completes using the tab character. For example, when I was typing in ":l myfunc1.hs" I actually just typed ":l myf" and then hit the tab character to complete the file name. Experiment with auto-completion, it will save you a lot of typing. In the following example, for instance, after defining the variable **sentence** I can just type "se" and the tab character to auto-complete the entire variable name:

```
*Main> let sentence = myfunc1 ["the", "cat"] ["ran", "up", "a", "tree"]
*Main> sentence
["the","cat","ran","up","a","tree"]
```

The function **head** returns the first element in a list and the function **tail** returns all but the first elements in a list:

```
*Main> head sentence
"the"
*Main> tail sentence
["cat","ran","up","a","tree"]
```

We can create new functions from existing arguments by supplying few arguments, a process known as "currying":

```
*Main> let p1 = (+ 1)
*Main> :t p1
p1 :: Num a => a -> a
*Main> p1 20
21
```

In this last example the function **+** takes two arguments but if we only supply one argument a function is returned as the value: in this case a function that adds 1 to an input value.

We can also create new functions by *composing* existing functions using the infix function . that when placed between two function names produces a new function that combines the two functions. Let's look at an example that uses . to combine the partial function **(+ 1)** with the function **length**:

```
*Main> let lengthp1 = (+ 1) . length
*Main> :t lengthp1
lengthp1 :: Foldable t => t a -> Int
*Main> lengthp1 "dog"
4
```

Note the order of the arguments to the inline function .: the argument on the right side is the first function that is applied, then the function on the left side of the . is applied.

This is the second example where we have seen the type **Foldable** which means that a type can be mapped over, or iterated over. We will look at Haskell types in the next section.

Introduction to Haskell Types

This is a good time to spend more time studying Haskell types. We will see more material on Haskell types throughout this book so this is just an introduction using the **data** expression to define a Type **MyColors** defined in the file **MyColors.hs**:

```
1 data MyColors = Orange | Red | Blue | Green | Silver
2  deriving (Show)
```

This example is incomplete so we will modify it soon. Line 1 defines the possible values for our new type **MyCOlors**. On line 2, we are asking the Haskell compiler to automatically generate a function **show** that can convert a value to a string. **show** is a standard function and in general we want it defined for all types. **show** converts an instance to a string value.

```
Prelude> :l colors.hs
[1 of 1] Compiling Main             ( colors.hs, interpreted )
Ok, modules loaded: Main.
*Main> show Red
"Red"
*Main> let c1 = Green
*Main> c1
Green
*Main> :t c1
c1 :: MyColors
*Main> Red == Green

<interactive>:60:5:
    No instance for (Eq MyColors) arising from a use of ‘==’
    In the expression: Red == Green
    In an equation for ‘it’: it = Red == Green
```

What went wrong here? The infix function == checks for equality and we did not define equality functions for our new type. Let's fix the definition in the file colors.hs:

```
data MyColors = Orange | Red | Blue | Green | Silver
 deriving (Show, Eq)
```

Because we are deriving **Eq** we are also asking the compiler to generate code to see if two instances of this class are equal. If we wanted to be able to order our colors then we would also derive **Ord**.

Now our new type has **show**, ==, and /= (inequality) defined:

```
Prelude> :l colors.hs
[1 of 1] Compiling Main             ( colors.hs, interpreted )
Ok, modules loaded: Main.
*Main> Red == Green
False
*Main> Red /= Green
True
```

Let's also now derive **Ord** to have the compile generate a default function **compare** that operates on the type **MyColors**:

```
data MyColors = Orange | Red | Blue | Green | Silver
 deriving (Show, Eq, Ord)
```

Because we are now deriving **Ord** the compiler will generate functions to calculate relative ordering for values of type **MyColors**. Let's experiment with this:

```
*Main> :l MyColors.hs
[1 of 1] Compiling Main             ( MyColors.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t compare
compare :: Ord a => a -> a -> Ordering
*Main> compare Green Blue
GT
*Main> compare Blue Green
LT
*Main> Orange < Red
True
*Main> Red < Orange
False
*Main> Green < Red
```

```
15 False
16 *Main> Green < Silver
17 True
18 *Main> Green > Red
19 True
```

Notice that the compiler generates a **compare** function for the type **MyColors** that orders values by the order that they appear in the **data do** expression. What if you wanted to order them in string sort order? This is very simple: we will remove **Ord** from the deriving clause and define our own function **compare** for type **MyColors** instead of letting the compiler generate it for us:

```
1 data MyColors = Orange | Red | Blue | Green | Silver
2  deriving (Show, Eq)
3
4 instance Ord MyColors where
5   compare c1 c2 = compare (show c1) (show c2)
```

In line 5 I am using the function **show** to convert instances of **MyColors** to strings and then the version of **compare** that is called in line 5 is the version the compiler wrote for us because we derived **Show**. Now the ordering is in string ascending sort order because we are using the **compare** function that is supplied for the type **String**:

```
1 *Main> :l MyColors.hs
2 [1 of 1] Compiling Main             ( MyColors.hs, interpreted )
3 Ok, modules loaded: Main.
4 *Main> Green > Red
5 False
```

Our new type MyColors is a simple type. Haskell also supports hierarchies of types called **Type Classes** and the type we have seen earlier **Foldable** is an example of a type class that other types can inherit from. For now, consider sub-types of **Foldable** to be collections like lists and trees that can be iterated over.

I want you to get in the habit of using :**type** and :**info** (usually abbreviated to :**t** and :**i**) in the GHCi repl. Stop reading for a minute now and type :**info Ord** in an open repl. You will get a lot of output showing you all of the types that **Ord** is defined for. Here is a small bit of what gets printed:

```
1  *Main> :i Ord
2  class Eq a => Ord a where
3    compare :: a -> a -> Ordering
4    (<) :: a -> a -> Bool
5    (<=) :: a -> a -> Bool
6    (>) :: a -> a -> Bool
7    (>=) :: a -> a -> Bool
8    max :: a -> a -> a
9    min :: a -> a -> a
10           -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
11 instance Ord MyColors -- Defined at MyColors.hs:4:10
12 instance (Ord a, Ord b) => Ord (Either a b)
13   -- Defined in ‘Data.Either’
14 instance Ord a => Ord [a]
15   -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
16 instance Ord Word -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
17 instance Ord Ordering -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
18 instance Ord Int -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
19 instance Ord Float -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
20 instance Ord Double -- Defined in ‘ghc-prim-0.4.0.0:GHC.Classes’
```

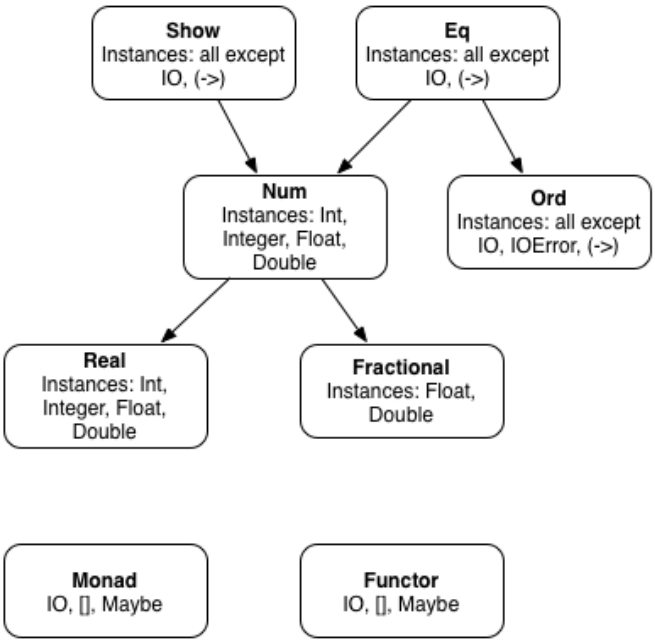
Lines 1 through 8 show you that **Ord** is a subtype of **Eq** that defines functions **compare**, **max**, and **min** as well as the four operators <, <=, >=, and >=. When we customized the **compare** function for the type **MyColors**, we only implemented **compare**. That is all that we needed to do since the other operators rely on the implementation of **compare**.

Once again, I ask you to experiment with the example type **MyColors** in an open GHCi repl:

```
1  *Main> :t max
2  max :: Ord a => a -> a -> a
3  *Main> :t Green
4  Green :: MyColors
5  *Main> :i Green
6  data MyColors = ... | Green | ...          -- Defined at MyColors.hs:1:39
7  *Main> max Green Red
8  Red
```

The following diagram shows a partial type hierarchy of a few types included in the standard Haskell Prelude (this is derived from the Haskell Report at haskell.org[11]):

[11] https://www.haskell.org/onlinereport/basic.html



Example Haskell Type Hierarchy

Here you see that type **Num** and **Ord** are sub-types of type **Eq**, **Real** is a sub-type of **Num**, etc. We will see the types **Monad** and **Functor** in the next chapter.

Functions Are Pure

Again, it is worth pointing out that Haskell functions do not modify their inputs values. The common pattern is to pass immutable values to a function and modified values are returned. As a first example of this pattern we will look at the standard function **map** that takes two arguments: a function that converts a value of any type **a** to another type **b**, and a list of type **a**. Functions that take other functions as arguments are called **higher order functions**. The result is another list of the same length whose elements are of type **b** and the elements are calulated using the function passed as the first argument. Let's look at a simple example using the function **(+ 1)** that adds 1 to a value:

```
1  *Main> :t map
2  map :: (a -> b) -> [a] -> [b]
3  *Main> map (+ 1) [10,20,30]
4  [11,21,31]
5  *Main> map (show . (+ 1)) [10,20,30]
6  ["11","21","31"]
```

In the first example, types **a** and **b** are the same, a **Num**. The second example used a composed function that adds 1 and then converts the example to a string. Remember: the function **show**

converts a Haskell data value to a string. In this second example types **a** and **b** are different because the function is mapping a number to a string.

The directory *haskell_tutorial_cookbook_examples/Pure* contains the examples for this chapter. We previously used the example file *Simple.hs*. Please note that in the rest of this book I will omit the git repository top level directory name haskell_tutorial_cookbook_examples and just specify the sub-directory name:

```
1 module Main where
2
3 sum2 x y = x + y
4
5 main = do
6   putStrLn ("1 + 2 = " ++ show (sum2 1 2))
```

For now let's just look at the mechanics of executing this file without using the REPL (started with *stack* **ghci**). We can simply build and run this example using *stack*, which is covered in some detail in Appendix A:

```
stack build --exec Simple
```

This command builds the project defined in the configuration files *Pure.cabal* and *stack.yaml* (the format and use of these files is briefly covered in detail in Appendix A and there is more reference material here[12]). This example defines two functions: **sum2** and **main**. **sum2** is a pure Haskell function with no state, no interaction with the outside world like file IO, etc., and no non-determinism. **main** is an impure function, and we will look at impure Haskell code in some detail in the next chapter. As you might guess the output of this code snippet is

```
1 + 2 = 3
```

To continue the tutorial on using pure Haskell functions, once again we will use *stack* to start an interactive repl during development:

[12] https://docs.haskellstack.org/en/stable/yaml_configuration/

```
markw@linux:~/haskell_tutorial_cookbook_examples/Pure$ stack ghci
*Main> :t 3
3 :: Num a => a
*Main> :t "dog"
"dog" :: [Char]
*Main> :t main
main :: IO ()
*Main>
```

In this last listing I don't show the information about your Haskell environment and the packages that were loaded. In repl listings in the remainder of this book I will continue to edit out this Haskell environment information for brevity.

Line 4 shows the use of the repl shortcut *:t* to print out the type of a string which is an array of [Char], and the type of the function **main** is of type **IO Action**, which we will explain in the next chapter. An IO action contains impure code where we can read and write files, perform a network operation, etc. and we will look at **IO Action** in the next chapter.

Using Parenthesis or the Special $ Character and Operator Precedence

We will look at operator and function precedence and the use of the **$** character to simplify using parenthesis in expessions. By the way, in Haskell there is not much difference between operators and function calls except operators like **+**, etc. which are by default infix while functions are usually prefix. So except for infix functions that are enclosed in backticks (e.g., **10 `div` 3**) Haskell usually uses prefix functions: a function followed by zero or more arguments. You can also use **$** that acts as an opening parenthesis with a not-shown closing parenthesis at the end of the current expression (which may be multi-line). Here are some examples:

```
*Main> print (3 * 2)
6
*Main> print $ 3 * 2
6
*Main> last (take 10 [1..])
10
*Main> last $ take 10 [1..]
10
*Main> ((take 10 [1..]) ++ (take 10 [1000..]))
[1,2,3,4,5,6,7,8,9,10,1000,1001,1002,1003,1004,1005,1006,1007,1008,1009]
*Main> take 10 [1..] ++ take 10 [1000..]
[1,2,3,4,5,6,7,8,9,10,1000,1001,1002,1003,1004,1005,1006,1007,1008,1009]
```

```
*Main> 1 + 2 * (4 * 5)
41
*Main> 2 * 3 + 10 * 30
306
```

I use the GHCi command **:info** (**:i** is an abbreviation) to check both operator precedence and the function signature if the operator is converted to a function by enclosing it in parenthessis:

```
*Main> :info *
class Num a where
  ...
  (*) :: a -> a -> a
  ...
          -- Defined in ‘GHC.Num’
infixl 7 *
*Main> :info +
class Num a where
  (+) :: a -> a -> a
  ...
          -- Defined in ‘GHC.Num’
infixl 6 +
*Main> :info `div`
class (Real a, Enum a) => Integral a where
  ...
  div :: a -> a -> a
  ...
          -- Defined in ‘GHC.Real’
infixl 7 `div`
*Main> :i +
class Num a where
  (+) :: a -> a -> a
  ...
          -- Defined in ‘GHC.Num’
infixl 6 +
```

Notice how **+** has lower precedence than *.

Just to be clear, understand how operators are used as functions and also how functions can be used as infix operators:

```
*Main> 2 * 3
6
*Main> (*) 2 3
6
*Main> 10 `div` 3
3
*Main> div 10 3
3
```

Especially when you are just starting to use Haskell it is a good idea to also use :**info** to check the type signatures of standard functions that you use. For example:

```
*Main> :info last
last :: [a] -> a         -- Defined in ‘GHC.List’
*Main> :info map
map :: (a -> b) -> [a] -> [b]         -- Defined in ‘GHC.Base’
```

Lazy Evaluation

Haskell is refered to as a *lazy language* because expressions are not evaluated until they are used. Consider the following example:

```
1  $ stack ghci
2  *Main> [0..10]
3  [0,1,2,3,4,5,6,7,8,9,10]
4  *Main> take 11 [0..]
5  [0,1,2,3,4,5,6,7,8,9,10]
6  *Main> let xs = [0..]
7  *Main> :sprint xs
8  xs = _
9  *Main> take 5 xs
10 [0,1,2,3,4]
11 *Main> :sprint xs
12 xs = _
13 *Main>
```

In line 2 we are creating a list with 11 elements. In line 4 we are doing two things:

- Creating an infinitely long list containing ascending integers starting with 0.
- Fetching the first 11 elements of this infinitely long list. It is important to understand that in line 4 only the first 11 elements are generated because that is all the **take** function requires.

In line 6 we are assigning another infinitely long list to the variable **xs** but the value of **xs** is unevaluated and a placeholder is stored to calculate values as required. In line 7 we use GHCi's :**sprint** command to show a value without evaluating it. The output in line 8 _ indicated that the expression has yet to be evaluated.

Lines 9 through 12 remind us that Haskell is a functional language: the **take** function used in line 9 does not change the value of its argument so **xs** as seen in lines 10 and 12 is still unevaluated.

Understanding List Comprehensions

Effectively using list comprehensions makes your code shorter, easier to understand, and easier to maintain. Let's start out with a few GHCi repl examples. You will learn a new GHCi repl trick in this section: entering multiple line expressions by using **:{** and **:}** to delay evaluation until an entire expression is entered in the repl (listings in this section are reformatted to fit the page width):

```
1  *Main> [x | x <- ["cat", "dog", "bird"]]
2  ["cat","dog","bird"]
3  *Main> :{
4  *Main| [(x,y) | x <- ["cat", "dog", "bird"],
5  *Main|          y <- [1..2]]
6  *Main| :}
7  [("cat",1),("cat",2),("dog",1),("dog",2),("bird",1),("bird",2)]
```

The list comprehension on line 1 assigns the elements of the list **["cat", "dog", "bird"]** one at a time to the variable **x** and then collects all these values of **x** in a list value that is the value of the list comprehension. The list comprehension in line 1 is hopefully easy to understand but when we bind and collect multiple variables the situation, as seen in the example in lines 4 and 5, is not as easy to understand. The thing to remember is that the first variable gets iterated as an "outer loop" and the second variable is iterated as the "inner loop." List comprehensions can use many variables and the iteration ordering rule is the same: last variable iterates first, etc.

```
*Main> :{
*Main| [(x,y) | x <- [0..3],
*Main|          y <- [1,3..10]]
*Main| :}
[(0,1),(0,3),(0,5),(0,7),(0,9),(1,1),(1,3),(1,5),(1,7),
 (1,9),(2,1),(2,3),(2,5),(2,7),(2,9),(3,1),(3,3),(3,5),
 (3,7),(3,9)]
*Main> [1,3..10]
[1,3,5,7,9]
```

In this last example we are generating all combinations of [0..3] and [1,3..10] and storing the combinations as two element tuples. You could also store then as lists:

```
*Main> [[x,y] | x <- [1,2], y <- [10,11]]
[[1,10],[1,11],[2,10],[2,11]]
```

List comprehensions can also contain filtering operations. Here is an example with one filter:

```
*Main> :{
*Main| [(x,y) | x <- ["cat", "dog", "bird"],
*Main|          y <- [1..10],
*Main|          y `mod` 3 == 0]
*Main| :}
[("cat",3),("cat",6),("cat",9),
 ("dog",3),("dog",6),("dog",9),
 ("bird",3),("bird",6),("bird",9)]
```

Here is a similar example with two filters (we are also filtering out all possible values of **x** that start with the character 'd'):

```
*Main> :{
*Main| [(x,y) | x <- ["cat", "dog", "bird"],
*Main|          y <- [1..10],
*Main|          y `mod` 3 == 0,
*Main|          x !! 0 /= 'd']
*Main| :}
[("cat",3),("cat",6),("cat",9),("bird",3),("bird",6),("bird",9)]
```

For simple filtering cases I usually use the **filter** function but list comprehensions are more versatile. List comprehensions are extremely useful - I use them frequently.

Lists are instances of the class **Monad** that we will cover in the next chapter (check out the section "List Comprehensions Using the **do** Notation").

List comprehensions are powerful. I would like to end this section with another trick that does not use list comprehensions for building lists of tuple values: using the **zip** function:

```
*Main> let animals = ["cat", "dog", "bird"]
*Main> zip [1..] animals
[(1,"cat"),(2,"dog"),(3,"bird")]
*Main> :info zip
zip :: [a] -> [b] -> [(a, b)]         -- Defined in 'GHC.List'
```

The function **zip** is often used in this way when we have a list of objects and we want to operate on the list while knowing the index of each element.

Haskell Rules for Indenting Code

When a line of code is indented relative to the previous line of code, or several lines of code with additional indentation, then the indented lines act as if they were on the previous line. In other words, if code that should all be on one line must be split to multiple lines, then use indentation as a signal to the Haskell compiler.

Indentation of continuation lines should be uniform, starting in the same column. Here are some examples of good code, and code that will not compile:

```
1   let a = 1     -- good
2       b = 2     -- good
3       c = 3     -- good
4
5   let
6     a = 1         -- good
7     b = 2         -- good
8     c = 3         -- good
9   in a + b + c    -- good
10
11  let a = 1     -- will not compile (bad)
12       b = 2    -- will not compile (bad)
13      c = 3     -- will not compile (bad)
14
15  let
16      a = 1     -- will not compile (bad)
17      b = 2     -- will not compile (bad)
18     c = 3      -- will not compile (bad)
19
20  let {
21      a = 1;     -- compiles but bad style (good)
22        b = 2;   -- compiles but bad style (good)
23      c = 3;     -- compiles but bad style (good)
24    }
```

If you use *C style* braces and semicolons to mark end of expressions, then indenting does not matter as seen in lines 20 through 24. Otherwise, uniform indentation is a hint to the compiler.

The same indenting rules apply to other types of **do** expressions which we will see throughout this book for **do**, **if**, and other types of **do** expressions.

Understanding let and where

At first glance, **let** and **where** seem very similar in that they allow us to create temporary variables used inside functions. As the examples in the file *LetAndWhere.hs* show, there are important differences.

In the following code notice that when we use **let** in pure code inside a function, we then use **in** to indicate the start of an expression to be evaluated that uses any variables defined in a **let** expression. Inside a **do** code block the **in** token is not needed and will cause a parse error if you use it. **do** code blocks are a syntactic sugar for use in impure Haskell code and we will use it frequently later in the book.

You also do not use **in** inside a list comprehension as seen in the function **testLetComprehension** in the next code listing:

```
1  module Main where
2
3  funnySummation w x y z =
4    let bob = w + x
5        sally = y + z
6    in bob + sally
7
8  testLetComprehension =
9    [(a,b) | a <- [0..5], let b = 10 * a]
10
11 testWhereBlocks a =
12   z * q
13     where
14       z = a + 2
15       q = 2
16
17 functionWithWhere n  =
18   (n + 1) * tenn
19   where
20     tenn = 10 * n
21
22 main = do
23   print $ funnySummation 1 2 3 4
24   let n = "Rigby"
25   print n
26   print testLetComprehension
27   print $ testWhereBlocks 11
28   print $ functionWithWhere 1
```

Compare the **let do** expressions starting on line 4 and 24. The first **let** occurs in pure code and uses **in** to define one or more **do** expressions using values bound in the **let**. In line 24 we are inside a monad, specifically using the **do** notation and here **let** is used to define pure values that can be used later in the **do do** expression.

Loading the last code example and running the **main** function produces the following output:

```
1  *Main> :l LetAndWhere.hs
2  [1 of 1] Compiling Main             ( LetAndWhere.hs, interpreted )
3  Ok, modules loaded: Main.
4  *Main> main
5  10
6  "Rigby"
7  [(0,0),(1,10),(2,20),(3,30),(4,40),(5,50)]
8  26
9  20
```

This output is self explanatory except for line 7 that is the result of calling **testLetComprehension** that retuns an example list comprehension **[(a,b)|a<-[0..5],letb=10*a]**

Conditional do Expressions and Anonymous Functions

The examples in the next three sub-sections can be found in *haskell_tutorial_cookbook_examples/Pure/Conditionals.hs*. You should read the following sub-sections with this file loaded (some GHCi repl output removed for brevity):

```
1  haskell_tutorial_cookbook_examples/Pure$ stack ghci
2  *Main> :l Conditionals.hs
3  [1 of 1] Compiling Main             ( Conditionals.hs, interpreted )
4  Ok, modules loaded: Main.
5  *Main>
```

Simple Pattern Matching

We previously used the built-in functions **head** that returns the first element of a list and **tail** that returns a list with the first element removed. We will define these functions ourselves using what is called wild card pattern matching. It is common to append the single quote character ‘ to built-in functions when we redefine them so we name our new functions **head**’ and **tail**’. Remember when we used destructuring to access elements of a tuple? Wild card pattern matching is similar:

```
head'(x:_)  = x
tail'(_:xs) = xs
```

The underscore character _ matches anything and ignores the matched value. Our **head** and **tail** definitions work as expected:

```
*Main> head' ["bird","dog","cat"]
"bird"
*Main> tail' [0,1,2,3,4,5]
[1,2,3,4,5]
*Main> :type head'
head' :: [t] -> t
*Main> :t tail'
tail' :: [t] -> [t]
```

Of course we frequently do not want to ignore matched values. Here is a contrived example that expects a list of numbers and doubles the value of each element. As for all of the examples in this chapter, the following function is *pure*: it can not modify its argument(s) and always returns the same value given the same input argument(s):

```
1 doubleList [] = []
2 doubleList (x:xs) = (* 2) x : doubleList xs
```

In line 1 we start by defining a pattern to match the empty list. It is necessary to define this *terminating* condition because we are using recursion in line 2 and eventually we reach the end of the input list and make the recursive call **doubleList []**. If you leave out line 1 you then will see a runtime error like "Non-exhaustive patterns in function doubleList." As a Haskell beginner you probably hate Haskell error messages and as you start to write your own functions in source files and load them into a GHCi repl or compile them, you will initially probably hate compilation error messages also. I ask you to take on faith a bit of advice: Haskell error messages and warnings will end up saving you a lot of effort getting your code to work properly. Try to develop the attitude "Great! The Haskell compiler is helping me!" when you see runtime errors and compiler errors.

In line 2 notice how I didn't need to use extra parenthesis because of the operator and function application precedence rules.

```
*Main> doubleList [0..5]
[0,2,4,6,8,10]
*Main> :t doubleList
doubleList :: Num t => [t] -> [t]
```

This function **doubleList** seems very unsatisfactory because it is so specific. What if we wanted to triple or quadruple the elements of a list? Do we want to write two new functions? You might think of adding an argument that is the multiplier like this:

```
bumpList n [] = []
bumpList n (x:xs) = n * x : bumpList n xs
```

is better, being more abstract and more general purpose. However, we will do much better.

Before generalizing the list manipuation process further, I would like to make a comment on coding style, specifically on not using unneeded parenthesis. In the last exmple defining **bumpList** if you have superfluous parenthesis like this:

```
bumpList n (x:xs) = (n * x) : bumpList (n xs)
```

then the code still works correctly and is fairly readable. I would like you to get in the habit of avoiding extra uneeded parenthesis and one tool for doing this is running **hlint** (installing **hlint** is covered in Appendix A) on your Haskell code. Using **hlint** source file will provide warnings/suggestions like this:

```
haskell_tutorial_cookbook_examples/Pure$ hlint Conditionals.hs
Conditionals.hs:7:21: Warning: Redundant bracket
Found:
  ((* 2) x) : doubleList (xs)
Why not:
  (* 2) x : doubleList (xs)

Conditionals.hs:7:43: Error: Redundant bracket
Found:
  (xs)
Why not:
  xs
```

hlint is not only a tool for improving your code but also for teaching you how to better program using Haskell. Please note that **hlint** provides other suggestions for *Conditionals.hs* that I am ignoring that mostly suggest that I replace our mapping operations with using the built-in **map** function and use functional composition. The sample code is specifically to show examples of pattern matching and is not as concise as it could be.

Are you satisfied with the generality of the function **bumpList**? I hope that you are not! We should write a function that will apply an arbitrary function to each element of a list. We will call this function **map'** to avoid confusing our **map'** function with the built-in function **map**.

The following is a simple implementation of a map function (we will see Haskell's standard map functions in the next section):

```
1  map' f [] = []
2  map' f (x:xs) = f x : map' f xs
```

In line 2 we do not need parenthesis around **f x** because function application has a higher precidence than the operator : which adds an element to the beginning of a list.

Are you pleased with how concise this definition of a map function is? Is concise code like **map'** readable to you? Speaking of someone who has written hundreds of thousands of lines of Java code for customers, let me tell you that I love the conciseness and readability of Haskell! I appreciate the Java ecosystem with many useful libraries and frameworks and augmented like fine languages like Clojure and JRuby, but in my opinion using Haskell is a more enjoyable and generally more productive language and programming environment.

Let's experiment with our **map'** function:

```
1  *Main> map' (* 7) [0..5]
2  [0,7,14,21,28,35]
3  *Main> map' (+ 1.1) [0..5]
4  [1.1,2.1,3.1,4.1,5.1,6.1]
5  *Main>  map' (\x -> (x + 1) * 2) [0..5]
6  [2,4,6,8,10,12]
```

Lines 1 and 3 should be understandable to you: we are creating a partial function like **(* 7)** and passing it to **map'** to apply to the list **[0..5]**.

The syntax for the function in line 5 is called an *anonymous function*. Lisp programers, like myself, refer to this as a lambda expression. In any case, I often prefer using *anonymous functions* when a function will not be used elsewhere. In line 5 the argement to the anonymous inline function is **x** and the body of the function is **(x + 1) * 2**.

I do ask you to not get carried away with using too many anonymous inline functions because they can make code a little less readable. When we put our code in modules, by default every symbol (like function names) in the module is externally visible. However, if we explicitly export symbols in a module **do** expression then only the explicitly exported symbols are visible by other code that uses the module. Here is an example:

```
module Test2 (doubler) where

map' f [] = []
map' f (x:xs) = (f x) : map' f xs

testFunc x = (x + 1) * 2

doubler xs = map' (* 2) xs
```

In this example **map'** and **testFunc** are hidden: any other module that imports **Test2** only has access to **doubler**. It might help for you to think of the exported functions roughly as an interface for a module.

Pattern Matching With Guards

We will cover two important concepts in this section: using guard pattern matching to make function definitions shorter and easier to read and we will look at the **Maybe** type and how it is used. The **Maybe** type is mostly used in non-pure Haskell code and we will use it heavily later. The Maybe type is a Monad (covered in the next chapter). I introduce the **Maybe** type here since its use fits naturally with guard patterns.

Guards are more flexible than the pattern matching seen in the last section. I use pattern matching for simple cases of destructuring data and guards when I need the flexibility. You may want to revisit the examples in the last section after experimenting with and understanding the examples seen here.

The examples for this section are in the file *Guards.hs*. As a first simple example we will implement the Ruby language "spaceship operator":

```
1  spaceship n
2    | n < 0      = -1
3    | n == 0     = 0
4    | otherwise = 1
```

Notice on line 1 that we do not use an = in the function definition when using guards. Each guard starts with |, contains a condition, and a value on the right side of the = sign.

```
1  *Main> spaceship (-10)
2  -1
3  *Main> spaceship 0
4  0
5  *Main> spaceship 17
6  1
```

Remember that a literal negative number as seen in line 1 must be wrapped in parenthesis, otherwise the Haskell compiler will interpret - as an operator.

Case Expressions

Case **do** expressions match a value against a list of possible values. It is common to use the wildcard matching value _ at the end of a case expression which can be of any type. Here is an example in the file *Cases.hs*:

```
1  module Main where
2
3  numberOpinion n =
4    case n of
5      0 -> "Too low"
6      1 -> "just right"
7      _ -> "OK, that is a number"
8
9  main = do
10   print $ numberOpinion 0
11   print $ numberOpinion 1
12   print $ numberOpinion 2
```

If Then Else expressions

Haskell has **if then else** syntax built into the language - **if** is not defined as a function. Personally I do not use **if then else** in Haskell very often. I mostly use simple pattern matching and guards. Here are some short examples from the file *IfThenElses.hs*:

```
ageToString age =
  if age < 21 then "minor" else "adult"
```

All **if** statements must have both a **then** expression and a **else** expression.

```
haskell_tutorial_cookbook_examples/Pure$ stack ghci
*Main> :l IfThenElses.hs
[1 of 1] Compiling Main             ( IfThenElses.hs, interpreted )
Ok, modules loaded: Main.
*Main> ageToString 15
"minor"
*Main> ageToString 37
"adult"
```

Maps

Maps are simple to construct using a list of key-value tuples and are by default immutable. There is an example using mutable maps in the next chapter.

We will look at the module **Data.Map** first in a GHCi repl, then later in a few full code examples. There is something new in line 1 of the following listing: I am assigning a short alias **M** to the module **Data.Map**. In referencing a function like **fromList** (which converts a list of tuples to a map) in the **Data.Map** module I can use **M.fromList** instead of **Data.Map.fromList**. This is a common practice so when you read someone else's Haskell code, one of the first things you should do when reading a Haskell source file is to make note of the module name abbreviations at the top of the file.

```
1  haskell_tutorial_cookbook_examples/Pure$ stack ghci
2  *Main> import qualified Data.Map as M
3  *Main M> :t M.fromList
4  M.fromList :: Ord k => [(k, a)] -> M.Map k a
5  *Main M> let aTestMap = M.fromList [("height", 120), ("weight", 15)]
6  *Main M> :t aTestMap
7  aTestMap :: Num a => M.Map [Char] a
8  *Main M> :t lookup
9  lookup :: Eq a => a -> [(a, b)] -> Maybe b
10 *Main M> :t M.lookup
11 M.lookup :: Ord k => k -> M.Map k a -> Maybe a
12 *Main M> M.lookup "weight" aTestMap
13 Just 15
14 *Main M> M.lookup "address" aTestMap
15 Nothing
```

The keys in a map must all be the same type and the values are also constrained to be of the same type. I almost always create maps using the helper function **fromList** in the module **Data.Maps**. We will only be using this method of map creation in later examples in this book so I am skipping coverage of other map building functions. I refer you to the Data.Map documentation[13].

The following example shows one way to use the **Just** and **Nothing** return values:

[13] https://www.stackage.org/haddock/lts-6.17/containers-0.5.6.2/Data-Map.html

```haskell
module MapExamples where

import qualified Data.Map as M -- from library containers

aTestMap = M.fromList [("height", 120), ("weight", 15)]

getNumericValue key aMap =
  case M.lookup key aMap of
    Nothing -> -1
    Just value -> value

main = do
  print $ getNumericValue "height" aTestMap
  print $ getNumericValue "age" aTestMap
```

The function **getNumericValue** shows one way to extract a value from an instance of type **Maybe**. The function **lookup** returns a **Maybe** value and in this example I use a **case** statement to test for a **Nothing** value or extract a wrapped value in a **Just** instance. Using **Maybe** in Haskell is a better alternative to checking for **null** values in *C* or *Java*.

The output from running the **main** function in module **MapExamples** is:

```
haskell_tutorial_cookbook_examples/Pure$ stack ghci
*Main> :l MapExamples.hs
[1 of 1] Compiling MapExamples      ( MapExamples.hs, interpreted )
Ok, modules loaded: MapExamples.
*MapExamples> main
120
-1
```

Sets

The documentation of **Data.Set.Class** can be found here[14] and contains overloaded functions for the types of sets defined here[15].

For most of my work and for the examples later in this book, I create immutable sets from lists and the only operation I perform is checking to see if a value is in the set. The following examples in GHCI repl are what you need for the material in this book:

[14] https://www.stackage.org/haddock/lts-6.17/sets-0.0.5/Data-Set-Class.html
[15] https://www.stackage.org/package/sets

```
*Main> import qualified Data.Set as S
*Main S> let testSet = S.fromList ["cat","dog","bird"]
*Main S> :t testSet
testSet :: S.Set [Char]
*Main S> S.member "bird" testSet
True
*Main S> S.member "snake" testSet
False
```

Sets and Maps are immutable so I find creating maps using a lists of key-value tuples and creating sets using lists is fine. That said, coming from the mutable Java, Ruby, Python, and Lisp programming languages, it took me a while to get used to immutability in Haskell.

More on Functions

In this section we will review what you have learned so far about Haskell functions and then look at a few more complex examples.

We have been defining and using simple functions and we have seen that operators behave like infix functions. We can make operators act as prefix functions by wrapping them in parenthesis:

```
*Main> 10 + 1
11
*Main> (+) 10 1
11
```

and we can make functions act as infix operators:

```
*Main> div 100 9
11
*Main> 100 `div` 9
11
```

This back tick function to operator syntax works with functions we write also:

```
*Main> let myAdd a b = a + b
*Main> :t myAdd
myAdd :: Num a => a -> a -> a
*Main> myAdd 1 2
3
*Main> 1 `myAdd` 2
3
```

Because we are working in a GHCi repl, in line 1 we use **let** to define the function **myAdd**. If you defined this function in a file and then loaded it, you would not use a **let**.

In the map examples where we applied a function to a list of values, so far we have used functions that map input values to the same return type, like this (using both partial function evaluation and anonymous inline function):

```
*Main> map (* 2) [5,6]
[10,12]
*Main> map (\x -> 2 * x) [5,6]
[10,12]
```

We can also map to different types; in this example we map from a list of **Num** values to a list containing sub-lists of **Num** values:

```
1 *Main> let makeList n = [0..n]
2 *Main> makeList 3
3 [0,1,2,3]
4 *Main> map makeList [2,3,4]
5 [[0,1,2],[0,1,2,3],[0,1,2,3,4]]
```

As usual, I recommend that when you work in a GHCi repl you check the types of functions and values you are working with:

```
1 *Main> :t makeList
2 makeList :: (Enum t, Num t) => t -> [t]
3 *Main> :t [1,2]
4 [1,2] :: Num t => [t]
5 *Main> :t [[0,1,2],[0,1,2,3],[0,1,2,3,4]]
6 [[0,1,2],[0,1,2,3],[0,1,2,3,4]] :: Num t => [[t]]
7 *Main>
```

In line 2 we see that for any type **t** the function signature is **t -> [t]** where the compiler determines that **t** is constrained to be a **Num** or **Enum** by examining how the input variable is used as a range parameter for contructing a list. Let's make a new function that works on any type:

```
1  *Main> let make3 x = [x,x,x]
2  *Main> :t make3
3  make3 :: t -> [t]
4  *Main> :t make3 "abc"
5  make3 "abc" :: [[Char]]
6  *Main> make3 "abc"
7  ["abc","abc","abc"]
8  *Main> make3 7.1
9  [7.1,7.1,7.1]
10 *Main> :t make3 7.1
11 make3 7.1 :: Fractional t => [t]
```

Notice in line 3 that the function **make3** takes any type of input and returns a list of elements the same type as the input. We used **makes3** both with a string argument and a fractional (floating point) number) argument.

Comments on Dealing With Immutable Data and How to Structure Programs

If you program in other programming languages that use mutable data then expect some feelings of disorientation initially when starting to use Haskell. It is common in other languages to maintain the state of a computation in an object and to mutate the value(s) in that object. While I cover mutable state in the next chapter the common pattern in Haskell is to create a data structure (we will use lists in examples here) and pass it to functions that return a new modified copy of the data structure as the returned value from the function. It is very common to keep passing the modified new copy of a data structure through a series of function calls. This may seem cumbersome when you are starting to use Haskell but quickly feels natural.

The following example shows a simple case where a list is constructed in the function **main** and passed through two functions **doubleOddElements** and **times10Elements**:

```
1  module ChainedCalls where
2
3  doubleOddElements =
4    map (\x -> if x `mod` 2 == 0 then x else 2 * x)
5
6  times10Elements = map (* 10)
7
8  main = do
9    print $ doubleOddElements [0,1,2,3,4,5,6,7,8]
10   let aList = [0,1,2,3,4,5]
```

```
11    let newList = times10Elements $ doubleOddElements aList
12    print newList
13    let newList2 = (times10Elements . doubleOddElements) aList
14    print newList2
```

Notice that the expressions being evaluated in lines 11 and 13 are the same. In line 11 we are applying function **doubleOddElements** to the value of **aList** and passing this value to the outer function **times10Elements**. In line 13 we are creating a new function from composing two existing functions: **times10Elements . doubleOddElements**. The parenthesis in line 13 are required because the . operator has lower precedence than the application of function **doubleOddElements** so without the parenthesis line 13 would evaluate as **times10Elements (doubleOddElements aList)** which is not what I intended and would throw an error.

The output is:

```
1  haskell_tutorial_cookbook_examples/Pure$ stack ghci
2  *Main> :l ChainedCalls.hs
3  [1 of 1] Compiling ChainedCalls     ( ChainedCalls.hs, interpreted )
4  Ok, modules loaded: ChainedCalls.
5  *ChainedCalls> main
6  [0,2,2,6,4,10,6,14,8]
7  [0,20,20,60,40,100]
8  [0,20,20,60,40,100]
```

Using immutable data takes some getting used to. I am going to digress for a minute to talk about working with Haskell. The steps I take when writing new Haskell code are:

- Be sure I understand the problem
- How will data be represented - in Haskell I prefer using built-in types when possible
- Determine which Haskell standard functions, modules, and 3rd party modules might be useful
- Write and test the pure Haskell functions I think that I need for the application
- Write an impure **main** function that fetches required data, calls the pure functions (which are no longer pure in the sense they are called from impure code), and saves the processed data.

I am showing you many tiny examples but please keep in mind the entire process of writing longer programs.

Error Handling

We have seen examples of handling soft errors when no value can be calculated: use **Maybe**, **Just**, and **Nothing**. In bug free pure Haskell code, runtime exceptions should be very rare and I usually do not try to trap them.

Using **Maybe**, **Just**, and **Nothing** is much better than, for example, throwing an error using the standard function **error**:

```
*Main> error "test error 123"
*** Exception: test error 123
```

and then, in impure code catching the errors, here is the documentation[16] for your reference.

In impure code that performs IO or accesses network resources that could possibly run out of memory, etc., runtime errors can occur and you could use the same **try catch** coding style that you have probably used in other programming languages. I admit this is my personal coding style but I don't like to catch runtime errors. I spent a long time writing Java applications and when possible I preferred using uncaught exceptions and I usually do the same when writing impure Haskell code.

Because of Haskell's type safety and excellent testing tools, it is possible to write nearly error free Haskell code. Later when we perform network IO we will rely on library support to handle errors and timeouts in a clean "Haskell like" way.

Testing Haskell Code

The example in this section is found in the directory *haskell_tutorial_cookbook_examples/Testing-Haskell*.

If you use *stack* to create a new project then the framework for testing is generated for you:

```
$ stack new TestingHaskell
$ cd TestingHaskell
$  ls -R
LICENSE                      app                          test
Setup.hs               src
TestingHaskell.cabal       stack.yaml

TestingHaskell//app:
Main.hs

TestingHaskell//src:
Lib.hs

TestingHaskell//test:
Spec.hs

$ cat test/Spec.hs
main :: IO ()
main = putStrLn "Test suite not yet implemented"
$ stack setup
$ stack build
```

[16] https://wiki.haskell.org/Exception

This *stack* generated project is more complex than the project I created manually in the directory *haskell_tutorial_cookbook_examples/Pure*. The file *Setup.hs* is a placeholder and uses any module named **Main** in the *app* directory. This module, defined in *app/Main.hs*, imports the module **Lib** defined in *src/Lib.hs*.

The generated test does not do anything, but let's run it anyway:

```
$ stack test
Registering TestingHaskell-0.1.0.0...
TestingHaskell-0.1.0.0: test (suite: TestingHaskell-test)

Progress: 1/2 Test suite not yet implemented

Completed 2 action(s).
```

In the generated project, I made a few changes:

- removed src/Lib.hs
- added src/MyColors.hs providing the type MyColors that we defined earlier
- modified app/Main.hs to use the MyColors type
- added tests to test/Spec.hs

Here is the contents of *TestingHaskell/src/MyColors.hs*:

```
module MyColors where

data MyColors = Orange | Red | Blue | Green | Silver
 deriving (Show, Eq)

instance Ord MyColors where
  compare c1 c2 = compare (show c1) (show c2)
```

And the new *test/Spec.hs* file:

```haskell
import Test.Hspec

import MyColors

main :: IO ()
main = hspec spec

spec :: Spec
spec = do
  describe "head" $ do
    it "test removing first list element" $ do
      head [1,2,3,4] `shouldBe` 1
      head ["the", "dog", "ran"] `shouldBe` "dog" -- should fail
  describe "MyColors tests" $ do
    it "test custom 'compare' function" $ do
      MyColors.Green < MyColors.Red `shouldBe` True
      Red > Silver `shouldBe` True                -- should fail
```

Notice how two of the tests are meant to fail as an example. Let's run the tests:

```
$ stack test
TestingHaskell-0.1.0.0: test (suite: TestingHaskell-test)

Progress: 1/2
head
  test removing first list element FAILED [1]
MyColors tests
  test custom 'compare' function FAILED [2]

Failures:

  test/Spec.hs:13:
  1) head test removing first list element
       expected: "dog"
        but got: "the"

  test/Spec.hs:17:
  2) MyColors tests test custom 'compare' function
       expected: True
        but got: False

Randomized with seed 1233887367
```

```
23
24 Finished in 0.0139 seconds
25 2 examples, 2 failures
26
27 Completed 2 action(s).
28 Test suite failure for package TestingHaskell-0.1.0.0
29     TestingHaskell-test:  exited with: ExitFailure 1
30 Logs printed to console
```

In line one with **stack test** we are asking *stack* to run app tests in the subdirectory *test*. All Haskell source files in subdirectory *test* are assumed to be test files. In the listing for file *test/Spec.hs* we have two tests that fail on purpose and you see the output for the failed tests at lines 12-15 and 17-20.

Because the Haskell compiler does such a good job at finding type errors I have fewer errors in my Haskell code compared to languages like Ruby and Common Lisp. As a result I find myself writing fewer tests for my Haskell code than I would write in other languages. Still, I recommend some tests for each of your projects; decide for yourself how much relative effort you want to put into writing tests.

Pure Haskell Wrap Up

I hope you are starting to get an appreciation for using composition of functions and higher order functions to enable us to compose programs from smaller pieces that can be joined together.

This composition is made easier when using pure functions that always return the same value when called with the same type of arguments.

We will continue to see examples of how lazy evaluation simplifies code because we can use infinitely large lists with the assurance that values are not calculated until they are needed.

In addition to Haskell code generally having fewer errors (after it gets by the compiler!) other advantages of functional programming include more concise code that is easy to read and understand once you get some experience with the language.

Tutorial on Impure Haskell Programming

One of the great things about Haskell is that the language encourages us to think of our code in two parts:

- Pure functional code (functions have no side effects) that is easy to write and test. Functional code tends to be shorter and less likely to be imperative (i.e., more functional, using maps and recursion, and less use of loops as in Java or C++).
- Impure code that deals with side effects like file and network IO, maintaining state in a typesafe way, and isolate imperative code that has side effects.

In his excellent functional programming with Haskell class at eDX[17] Erik Meijer described pure code as being islands in the ocean and the ocean representing impure code. He says that it is a design decision how much of your code is pure (islands) and how much is impure (the ocean). This model of looking at Haskel programs works for me.

My use the word "impure" is common for refering to Haskell code with side effects. Haskell is a purely functional language and side effects like I/O are best handled in a pure functional way using .

In addition to showing you reusable examples of impure code that you will likely need in your own programs, a major theme of this chapter is handling impure code in a convenient type safe fashion. Any **Monad**, which wraps a single value, is used to safely manage state. I will introduce you to using **Monad** types as required for the examples in this chapter. This tutorial style introduction will prepare you for understanding the sample applications later.

Hello IO () Monad

I showed you many examples of pure code in the last chapter but most examples in source files (as opposed to those shown in a GHCi repl) had a bit of impure code in them: the **main** function like the following that simply writes a string of characters to standard output:

[17] http://edx.org

```
main = do
  print "hello world"
```

The type of function **main** is:

```
*Main> :t main
main :: IO ()
```

The **IO ()** monad is an IO value wrapped in a type safe way. Because Haskell is a lazy evauation language, the value is not evaluated until it is used. Every **IO ()** action returns exactly one value. Think of the word “mono” (or “one”) when you think of Monads because they always return one value. Monads are also used to connnect together parts of a program.

What is it about the function **main** in the last example that makes its type an **IO ()**? Consider the simple **main** function here:

```
module NoIO where

main = do
  let i = 1 in
    2 * i
```

and its type:

```
*Main> :l NoIO
[1 of 1] Compiling NoIO             ( NoIO.hs, interpreted )
Ok, modules loaded: NoIO.
*NoIO> main
2
*NoIO> :t main
main :: Integer
*NoIO>
```

OK, now you see that there is nothing special about a **main** function: it gets its type from the type of value returned from the function. It is common to have the return type depend on the function argument types. The first example returns a type **IO ()** because it returns a print **do** expression:

```
*Main> :t print
print :: Show a => a -> IO ()
*Main> :t putStrLn
putStrLn :: String -> IO ()
```

The function **print** shows the enclosing quote characters when displaying a string while **putStrLn** does not. In the first example, what happens when we stitch together several expressions that have type **IO ()**? Consider:

```
main = do
  print 1
  print "cat"
```

Function **main** is still of type **IO ()**. You have seen **do** expressions frequently in examples and now we will dig into what the **do** expression is and why we use it.

The **do** notation makes working with monads easier. There are alternatives to using **do** that we will look at later.

One thing to note is that if you are doing bindings inside a **do** expression using a **let** with a **in** expression, you need to wrap the bindings in a new (inner) **do** expression if there is more than one line of code following the **let** statement. The way to avoid requiring a nested **do** expression is to not use **in** in a **let** expression inside a **do** block of code. Yes, this sounds complicated but let's clear up any confusion by looking at the examples found in the file *ImPure/DoLetExample.hs* (you might also want to look at the similar example file *ImPure/DoLetExample2.hs* that uses *bind* operators instead of a **do** statement; we will look at *bind* operators in the next section):

```
module DoLetExample where

example1 = do  -- good style
  putStrLn "Enter an integer number:"
  s <- getLine
  let number = (read s :: Int) + 2
  putStrLn $ "Number plus 2 = " ++ (show number)

example2 = do  -- avoid using "in" inside a do statement
  putStrLn "Enter an integer number:"
  s <- getLine
  let number = (read s :: Int) + 2 in
    putStrLn $ "Number plus 2 = " ++ (show number)

example3 = do  -- avoid using "in" inside a do statement
  putStrLn "Enter an integer number:"
```

```
  s <- getLine
  let number = (read s :: Int) + 2 in
    do -- this do is required since we have two dependent statements:
      putStrLn "Result is:"
      putStrLn $ "Number plus 2 = " ++ (show number)

main = do
  example1
  example2
  example3
```

You should use the pattern in function **example1** and not the pattern in **example2**. The **do** expression is syntactic sugar that allows programmers to string together a sequence of operations that can mix pure and impure code.

To be clear, the left arrow <- is used when the expression on the right side is some type of **IO ()** that needs to be *lifted* before being used. A **let do** expression is used when the right side expression is a pure value.

On lines 6 and 12 we are using function **read** to converting a string read out of **IO String ()** to an integer value. Remember that the value of **s** (from calling **readLine**) is an **IO ()** so in the same way you might read from a file, in this example we are reading a value from an **IO ()** value.

A Note About >> and >>= Operators

So far in this book I have been using the syntactic sugar of the **do** expression to work with Monads like **IO ()** and I will usually use this syntactic sugar for the rest of this book.

Even though I find it easier to write and read code using **do**, many Haskell programmers prefer >> and >>= so let's go over these operators so you won't be confused when reading Haskell code that uses them. Also, we use **do** expressions in code the compiler generates similar code using these >> and >>= operators.

The Monad type class defines the operators >>= and **return**. We turn to the GHCi repl to experiment with and learn about these operators:

```
1  *Main> :t (>>)
2  (>>) :: Monad m => m a -> m b -> m b
3  *Main> :t (>>=)
4  (>>=) :: Monad m => m a -> (a -> m b) -> m b
5  *Main> :t return
6  return :: Monad m => a -> m a
```

We start with the **return** function type **return** :: **Monad m** ⇒ **a -> m a** which tells us that for a monad **m** the function **return** takes a value and wraps it in a monad. We will see examples of the **return** function used to return a wrapped value from a function that returns **IO ()** values. The *bind* operator **(>>)** is used to evaluate two expressions in sequence. As an example, we can replace this **do** expression:

```
main = do
  example1
  example2
  example3
```

with the following:

```
main = example1 >> example2 >> example3
```

The operator >>= is similar to >> except that it evaluates the left hand expression and pipes its value into the right hand side expression. The left hand side expression is evaluated to some type of **IO ()** and the expression on the right hand side typically reads from the input **IO ()**. An example will make this simpler to understand:

```
1  module DoLetExample3 where
2
3  example3 =  putStrLn "Enter an integer number:" >>  getLine
4
5  example4 mv = do
6    let number = (read mv :: Int) + 2
7    putStrLn $ "Number plus 2 = " ++ (show number)
8
9  main = example3 >>= example4
```

Note that I could have used a **do** statement to define function **example3** but used a *bind* operator instead. Let's run this example and look at the function types. Please don't just quickly read through the following listing; when you understand what is happening in this example then for the rest of your life programming in Haskell things will be easier for you:

```
1  *DoLetExample3> main
2  Enter an integer number:
3  1
4  Number plus 2 = 3
5  *DoLetExample3> :t example3
6  example3 :: IO String
7  *DoLetExample3> :t example4
8  example4 :: String -> IO ()
9  *DoLetExample3> :t main
10 main :: IO ()
11 *DoLetExample3> let x = example3
12 *DoLetExample3> x
13 Enter an integer number:
14 4
15 "4"
16 *DoLetExample3> :t x
17 x :: IO String
18 *DoLetExample3> x >>= example4
19 Enter an integer number:
20 3
21 Number plus 2 = 5
```

The interesting part starts at line 11 when we define **x** to be the returned value from calling **example3**. Remember that Haskell is a lazy language: evaluation is postponed until a value is actually used.

Working inside a GHCi repl is like working interactively inside a **do** expression. When we evaluate **x** in line 12 then the code in function **example3** is actually executed (notice this is where the user prompt to enter a number occurs). In line 18 we are re-evaluationg the value in **x** and passing the resulting **IO String ()** value to the function **example4**.

Haskell is a "piecemeal" programming language as are the Lisp family of languages where a repl is used to write little pieces code that are collected into programs. For simple code in Haskell (and Lisp languages) I do sometimes directly enter code into a text editor but very ofter I start in a repl, experiment, debug, refine, and then copy into an edited file.

Console IO Example with Stack Configuration

The directory *CommandLineApps* contains two simple applications that interact with STDIO, that is to write to the console and read from the keyboard. The first example can be found in file *CommandLineApp/CommandLine1.hs*:

```
1  module Main where
2
3  import System.IO
4  import Data.Char (toUpper)
5
6  main = do
7    putStrLn "Enter a line of text for test 1:"
8    s <- getLine
9    putStrLn $ "As upper case:\t" ++ (map toUpper s)
10   main
```

Lines 3 and 4 import the entire **System.IO** module (that is, import all exported symbols from **System.IO**) and just the function **toUpper** from module **Data.Char**. **System.IO** is a standard Haskell module and we do not have to do anything special to import it. The **Data.Char** is stored in the package **text**. The package **text** is contained in the library package **base** which is specified in the *CommandLineApp.cabal* configuration file that we will look at soon.

Use of the <- assignment in line 8 in the last Haskell listing is important to understand. It might occur to you to leave out line 8 and just place the **getLine** function call directly in line 9, like this:

```
1    putStrLn $ "As upper case:\t" ++ (map toUpper getLine)
```

If you try this (please do!) you will see compilation errors like:

```
1    Couldn't match expected type ‘[Char]’ with actual type ‘IO String’
2    In the second argument of ‘map’, namely ‘getLine’
3    In the second argument of ‘(++)’, namely ‘(map toUpper getLine)’
```

The type of **getLine** is an **IO ()** that is a wrapped IO call. The value is not computed until it is used. The <- assignment in line 8 evaluates the IO call and unwraps the result of the IO operation so that it can be used.

I don’t spend much time covering *stack* project configuration files in this book but I do recommend that as you work through examples to also in each example directory look for a file ending with the file extension *.cabal* that specified which packages need to be loaded. For some examples it might take a while to download and configure libraries the first time you run either *stack build* or *stack ghci* in an example directory.

The Haskell stack project in the **CommandLineApp** directory has five target applications as we can see in the **CommandLineApp.cabal** file. I am not going to go into much detail about the project cabal and stack.yaml files generated by stack when you create a new project except for configuration data that I had to add manually; in this case, I added two executable targets at the end of the cabal file (note: the project in the github repository for this book has more executable targets, I just show a few here):

```
executable CommandLine1
  hs-source-dirs:      .
  main-is:             CommandLine1.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5

executable CommandLine2
  hs-source-dirs:      .
  main-is:             CommandLine2.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5

executable ReadTextFile
  hs-source-dirs:      .
  main-is:             ReadTextFile.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5

executable GameLoop1
  hs-source-dirs:      .
  main-is:             GameLoop1.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5, time

executable GameLoop2
  hs-source-dirs:      .
  main-is:             GameLoop2.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5, random
```

The executable name determines the compiled and linked executable file name. For line 1, an executable file "CommandLine1" (or "CommandLine1.exe"" on Windows) will be generated. The parameter **hs-source-dirs** is a comma separated list of source file directories. In this simple example all Haskell source files are in the project's top level directory .. The **build-depends** is a comma separated list of module libraries; here we only use the base built-in modules packaged with Haskell.

Let's use a GHCi repl to poke at this code and understand it better. The project defined in *CommandLineApp/CommandLineApp.cabal* contains many executable targets so when we enter a GHCi repl, the available targets are shown and you can choose one; in this case I am selecting the first target defined in the *cabal* file. In later GHCi repl listings, I will edit out this output for brevity:

```
$ stack ghci

* * * * * * * * *
The main module to load is ambiguous. Candidates are:
1. Package `CommandLineApp' component exe:CommandLine1 with main-is file: /Users\
/markw/GITHUB/haskell_tutorial_cookbook_examples/CommandLineApp/CommandLine1.hs
2. Package `CommandLineApp' component exe:CommandLine2 with main-is file: /Users\
/markw/GITHUB/haskell_tutorial_cookbook_examples/CommandLineApp/CommandLine2.hs
3. Package `CommandLineApp' component exe:ReadTextFile with main-is file: /Users\
/markw/GITHUB/haskell_tutorial_cookbook_examples/CommandLineApp/ReadTextFile.hs
You can specify which one to pick by:
 * Specifying targets to stack ghci e.g. stack ghci CommandLineApp:exe:CommandLi\
ne1
 * Specifying what the main is e.g. stack ghci --main-is CommandLineApp:exe:Comm\
andLine1
 * Choosing from the candidate above [1..3]
* * * * * * * * *

Specify main module to use (press enter to load none): 1
Loading main module from cadidate 1, --main-is /Users/markw/GITHUB/haskell_tutor\
ial_cookbook_examples/CommandLineApp/CommandLine1.hs

Configuring GHCi with the following packages: CommandLineApp
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Ok, modules loaded: none.
[1 of 1] Compiling Main             ( /Users/markw/GITHUB/haskell_tutorial_cookb\
ook_examples/CommandLineApp/CommandLine1.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t main
main :: IO b
*Main> :info main
main :: IO b
-- Defined at /Users/markw/GITHUB/haskell_tutorial_cookbook_examples/CommandLine\
App/CommandLine1.hs:6:1
*Main> :t getLine
getLine :: IO String
*Main> :t putStrLn
putStrLn :: String -> IO ()
*Main> main
Enter a line of text for test 1:
line 1
As upper case:        LINE 1
```

```
Enter a line of text for test 1:
line 2
As upper case:        LINE 2
Enter a line of text for test 1:
^C Interrupted.
*Main>
```

In line 36 the function **getLine** is of type **getLine :: IO String** which means that calling **getLine** returns a value that is a computation to get a line of text from *stdio* but the IO operation is not performed until the value is used.

Please note that it is unusual to put five executable targets in a project's *cabal* file. I am only doing so here because I wanted to group five similar examples together in this subdirectory of the github repo for this book[18]. This repo has 16 example subdirectories, and the number would be much greater if I didn't collect similar examples together.

We will use the example in file *CommandLine2.hs* in the next section which is similar to this example but also appends the user input to a text file.

File IO

We will now look at a short example of doing file IO. We will write Haskell simple string values to a file. If you are using the more efficient Haskell Text values, the code is the same. Text values are more efficient than simple string values when dealing with a lot of data and we will later use a compiler setting to automatically convert between the underlying formats. The following listing shows *CommandLineApp/CommandLine2.hs*:

```
1  module Main where
2
3  import System.IO
4  import Data.Char (toUpper)
5
6  main = do
7    putStrLn "Enter a line of text for test2:"
8    s <- getLine
9    putStrLn $ "As upper case:\t" ++ (map toUpper s)
10   appendFile "temp.txt" $ s ++ "\n"
11   main
```

Note the use of recursion in line 11 to make this program loop forever until you use a *COntrol-c* to stop the program.

[18] https://github.com/mark-watson/haskell_tutorial_cookbook_examples

In line 10 we are using function **appendFile** to open a file, append a string to it, and then close the file. **appendFile** is of type **appendFile :: FilePath -> String -> IO ()**. It looks like we are passing a simple string as a file name instead of type **FilePath** but if you look up the definition of **FilePath** you will see that it is just an alias for string: **type FilePath = String**.

Running this example in a GHCi repl, with much of the initial printout from running *stack ghci* not shown:

```
1  $ stack ghci
2  CommandLineApp-0.1.0.0: configure
3  Specify main module to use (press enter to load none): 2
4  Ok, modules loaded: Main.
5  *Main> main
6  Enter a line of text for test2:
7  line 1
8  As upper case:        LINE 1
9  Enter a line of text for test2:
10 line 2
11 As upper case:        LINE 2
12 Enter a line of text for test2:
13 ^C Interrupted.
14 *Main>
```

The file *temp.txt* was just created.

The next example used *ReadTextFile.hs* to read the file *temp.txt* and process the text by finding all words in the file:

```
1  module Main where
2
3  import System.IO
4  import Control.Monad
5
6  main = do
7    entireFileAsString <- readFile "temp.txt"
8    print entireFileAsString
9    let allWords = words entireFileAsString
10   print allWords
```

readFile is a high-level function because it manages for you reading a file and closing the file handle it uses internally. The built in function **words** splits a string on spaces and returns a list of strings **[String]** that are printed on line 7:

```
$ stack ghci
CommandLineApp-0.1.0.0: build
Specify main module to use (press enter to load none): 3
Ok, modules loaded: ReadTextFile.
*ReadTextFile> main
"line 1\nline 2\n"
["line","1","line","2"]
*ReadTextFile>
*ReadTextFile> :t readFile
readFile :: FilePath -> IO String
*ReadTextFile> :type words
words :: String -> [String]
```

What if the function **readFile** encounters an error? That is the subject for the next section.

Error Handling in Impure Code

I know you have been patiently waiting to see how we handle errors in Haskell code. Your wait is over! We will look at several common types of runtime errors and how to deal with them. In the last section we used the function **readFile** to read the contents of a text file *temp.txt*. What if *temp.txt* does not exist? Well, then we get an error like the following when running the example program in *ReadTextFile.hs*:

```
*Main> main
*** Exception: temp.txt: openFile: does not exist (No such file or directory)
```

Let's modify this last example in a new file *ReadTextFileErrorHandling.hs* that catches a file not found error. The following example is derived from the first example in Michael Snoyman's article Catching all exceptions[19]. This example does not work inside threads; if you need to catch errors inside a thread then see the second example in Michael's article.

[19] https://www.schoolofhaskell.com/user/snoyberg/general-haskell/exceptions/catching-all-exceptions

```
module Main where

import System.IO
import Control.Exception

-- catchAny by Michael Snoyman:
catchAny :: IO a -> (SomeException -> IO a) -> IO a
catchAny = Control.Exception.catch

safeFileReader :: FilePath -> IO String
safeFileReader fPath = do
  entireFileAsString <- catchAny (readFile "temp.txt") $ \error -> do
    putStrLn $ "Error: " ++ show error
    return ""
  return entireFileAsString

main :: IO ()
main = do
  fContents <- safeFileReader "temp.txt"
  print fContents
  print $ words fContents
```

I will run this twice: the first time without the file *temp.txt* present and a second time with *temp.txt* in the current durectory:

```
*Main> :l ReadTextFileErrorHandling.hs
[1 of 1] Compiling Main             ( ReadTextFileErrorHandling.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Error: temp.txt: openFile: does not exist (No such file or directory)
""
[]
1
*Main> main
"line 1\nline 2\n"
["line","1","line","2"]
```

Until you need to handle runtime errors in a multi-threaded Haskell program, following this example should be sufficient. In the next section we look at Network IO.

Network IO

We will experiment with three network IO examples in this book:

- A simple socket client/server example in this section.
- Reading web pages in the chapter "Web Scraping"
- Querying remote RDF endpoints in the chapter "Linked Data and the Semantic Web"

We start by using a high level library, **network-simple** for both the client and serve examples in the next two sub-sections. The client and sever examples are in the directory *haskell_tutorial_cookbook_examples/ClientServer* in the files *Client.hs* and *Server.hs*.

Server Using network-simple Library

The Haskell **Network** and **Network.Simple** modules use strings represented as **Data.ByteString.Char8** data so as seen in line 1 I set the language type *OverloadedStrings*. The following example in file *ClientServer/Server.hs* is derived from an example in the *network-simple* project:

```
1  {-# LANGUAGE OverloadedStrings #-}
2
3  module Server where
4
5  import Control.Monad
6  import qualified Data.ByteString.Char8 as B
7  import qualified Network.Simple.TCP as T
8
9  reverseStringLoop sock = do
10   mbs <- T.recv sock 4096
11   case mbs of
12     Just bs -> T.send sock (B.reverse bs) >> reverseStringLoop sock
13     Nothing -> return ()
14
15 main :: IO ()
16 main = T.withSocketsDo $ do -- derived from library example
17   T.listen "*" "3000" $ \(lsock, laddr) -> do
18     putStrLn $ "Listening at " ++ show laddr
19     forever . T.acceptFork lsock $ \(sock, addr) -> do
20       putStrLn $ "Connection from " ++ show addr
21       reverseStringLoop sock
```

The server accepts a string, reverses the string, and returns the reversed string to the client.

I am assuming that you have done some network programming and are familiar with sockets, etc. The function **reverseStringLoop** defined in lines 9-13 accepts a socket as a parameter and returns a value of type **MonadIO** that wraps a byte-string value. In line 10 we use the **T.recv** function that takes a socket and the maximum number of bytes to received from the client. The **case** expression

reverses the received byte string, sends the reversed string back to the client, and recursively calls itself waiting for new data from the client. If the client breaks the socket connection, then the function retuns an empty **MonadIO()**.

The **main** function defined in lines 15-21 listens on port 3000 for new client socket connections. In line 19, the function **T.acceptFork** accepts as an argument a socket value and a function to execute; the complete type is:

```
1  *Main> :t T.acceptFork
2  T.acceptFork
3    :: transformers-0.4.2.0:Control.Monad.IO.Class.MonadIO m =>
4       T.Socket
5       -> ((T.Socket, T.SockAddr) -> IO ()) -> m GHC.Conc.Sync.ThreadId
```

Don't let line 3 scare you; the GHCi repl is just showing you where this type of **MonadIO** is defined. The return type refers to a thread ID that is passed to the function **forever** :: **Monad m** ⇒ **m a -> m b** that is defined in the module **Control.Monad** and lets the thread run until it teminates.

The *network-simple* package is fairly high level and relatively simple to use. If you are interested you can find many client/server examples on the web that use the lower-level *network* package.

We will develop a client application to talk with this server in the next section but if you want to immediately try the server, start it and then run *telnet* in another terminal window:

```
Prelude> :l Server
[1 of 1] Compiling Server           ( Server.hs, interpreted )
Ok, modules loaded: Server.
*Main> main
Listening at 0.0.0.0:3000
```

And run *telnet*:

```
$ telnet localhost 3000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
12345
54321
The dog ran down the street
teerts eht nwod nar god ehT
```

In the next section we write a simple client to talk with this service example.

Client Using network-simple Library

I want to use automatic conversion between strings represented as **Data.ByteString.Char8** data and regular **[Char]** strings so as seen in line 1 I set the language type *OverloadedStrings* in the example in file *Client.hs*:

```
1  {-# LANGUAGE OverloadedStrings #-}
2
3  module Client where
4
5  import Control.Monad
6  import qualified Network.Simple.TCP as T
7
8  main = do
9    T.connect "127.0.0.1" "3000" $ \(connectionSocket, remoteAddr) -> do
10   putStrLn $ "Connection established to " ++ show remoteAddr
11   T.send connectionSocket "test123"
12   response <- T.recv connectionSocket 100
13   case response of
14     Just s -> putStrLn $ "Response: " ++ show s
15     Nothing -> putStrLn "No response from server"
```

The function **T.connect** in line 9 accepts arguments for a host name, a port, and a function to call with the connection socket to the server and the server's address. The body of this inline function, defined in in the middle on line 9 and continuing in lines 10-15, prints the server address, sends a string "test123" to the server, and waits for a response back from the server (**T.recv** in line 12). The server response is printed, or a warning that no response was received.

While the example in file *Server.hs* is running in another terminal, we can run the client interactively:

```
Prelude> :l Client.hs
[1 of 1] Compiling Client           ( Client.hs, interpreted )
Ok, modules loaded: Client.
*Main main
Connection established to 127.0.0.1:3000
Response: "321tset"
```

A Haskell Game Loop that Maintains State Functionally

The example in this section can be found in the file *GameLoop2.hs* in the directory *haskell_tutorial_cookbook_examples/CommandLineApp*. This example uses the random package to generate

a seed random number for a simple number guessing game. An alternative implementation in *GameLoop1.hs*, which I won't discuss, uses the system time to generate a seed.

This is an important example because it demonstrates one way to maintain state in a functional way. We have a read-only game state value that is passed to the function **gameLoop** which modifies the read-only game state passed as an argument and returns a newly constructed game state as the function's returned value. This is a common pattern that we will see again later when we develop an application to play a simplified version of the card game Blackjack in the chapter "Haskell Program to Play the Blackjack Card Game."

```
1  module GameLoop2 where
2
3  import System.Random
4
5  data GameState = GameState { numberToGuess::Integer, numTries::Integer}
6                   deriving (Show)
7
8  gameLoop :: GameState -> IO GameState
9  gameLoop gs = do
10   print $ numberToGuess gs
11   putStrLn "Enter a number:"
12   s <- getLine
13   let num = read s :: Integer
14   if num == numberToGuess gs then
15     return gs
16   else gameLoop $ GameState (numberToGuess gs) ((numTries gs) + 1)
17
18 main = do
19   pTime <- randomRIO(1,4)
20   let gameState = GameState pTime 1
21   print "Guess a number between 1 and 4"
22   gameLoop gameState
```

You notice in line 12 that since we are inside of a **do** expression we can *lift* (or unwrap) the **IO String ()** value returned from **getLine** to a string value that we can use directly. This is a pattern we will use repeatedly. The value returned from **getLine** is not used until line 13 when we use function **read** to extract the value from the **IO String ()** value **getLine** returned.

In the **if** expression in lines 14-16 we check if the user has input the correct value and can then simply return the input game state to the calling **main** function. If the user has not guessed the correct number then in line 16 we create a new game state value and call the function **gameLoop** recursively with the newly constructed game state.

The following listing shows a sample session playing the number guessing game.

```
Prelude> :l GameLoop2.hs
[1 of 1] Compiling GameLoop2        ( GameLoop2.hs, interpreted )
Ok, modules loaded: GameLoop2.
*GameLoop2> main
"Guess a number between 1 and 4"
Enter a number:
1
Enter a number:
3
Enter a number:
4
GameState {numberToGuess = 4, numTries = 3}
*GameLoop2> main
"Guess a number between 1 and 4"
Enter a number:
1
Enter a number:
2
GameState {numberToGuess = 2, numTries = 2}
*GameLoop2>
```

We will use this pattern for maintaining state in a game in the later chapter "Haskell Program to Play the Blackjack Card Game."

Efficiency of Haskell Strings

Except for the Client/Server example, so far we have been mostly using simple **String** values where **String** is a list of characters **[Char]**. For longer strings it is much more efficient to use the module **Data.Text**[20] that is defined in package **text** (so **text** needs to be added to the dependencies in your cabal file).

Many Haskell libraries use the simple **String** type but the use of **Data.Text** is also common, especially in applications handling large amounts of string data. We have already seen examples of this in the client/server example programs. Fortunately Haskell is a strongly typed language that supports a language extension for automatically handling both simple strings and the more efficient text types; this language extension, as we have seen in a previous example, is activated by adding the following near the top of a Haskell source file:

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

[20] https://www.stackage.org/nightly-2016-09-18/package/text-1.2.2.1

As much as possible I am going to use simple strings in this book and when we need both simple strings and byte strings I will then use *OverloadedStrings* for automatic conversion. This conversion is performed by knowing the type signatures of data and functions in surrounding code. The compiler figures out what type of string is expected and does the conversion for you.

A More Detailed Look at Monads

We have been casually using different types of **IO ()** monads. In this section I will introduce you to the **State** monad and then we will take a deeper look at **IO ()**. While we will be just skimming the surface of the topic of monads, my goal in this section is to teach you enough to work through the remaining examples in this book.

Monads are types belonging to the Monad type class that specifies one operator and one function:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

The >>= operator takes two arguments: a monad wrapping a value (type **a** in the above listing) and a function taking the same type **a** and returning a monad wrapping a new type **b**. The return value of >>= is a new monad wrapping a value of type **b**.

The Monad type class function **return** takes any value and wraps it in a new monad. The naming of **return** is confusing because it does not alter the flow of execution in a program like a *return* statement in Java, rather, it wraps a value in a monad.

State Monad

The definition for the constructor of a State monad is:

```
newtype State s a = State { runState :: s -> (a, s) }
```

So far we have been using **data** to define new types and **newtype** is similar except **newtype** acts during compile time and no type information is present at runtime. All monads conatin a value and for the State monad this value is a function. The >>= operator is called the *bind* operator.

The accessor function **runState** provides the means to access the value in the state. The following example is in the file *StateMonad/State1.hs*. In this example, **incrementState** is a state monad that increases its wrapped integer value by one when it is executed. Remember that the **return** function is perhaps poorly named because it does not immediately "return" from a computation block as it does in other languages; **return** simply wraps a value as a monad without redirecting the execution flow.

In order to make the following example more clear, I implement the increment state function twice, once using the **do** notation that you are already familiar with and once using the >>= bind operator:

```
1  module Main where
2
3  import Control.Monad.State
4
5  incrementState :: State Int Int
6  incrementState = do
7    n <- get
8    put (n + 1)
9    return n
10
11 -- same state monad without using a 'do' expression:
12 incrementState2 :: State Int Int
13 incrementState2 = get >>= \a ->
14                   put (a + 1) >>= \b ->
15                   return a
16
17 bumpVals (a,b) = (a+1, b+2)
18
19 main = do
20   print $ runState incrementState 1  -- (1,2) == (return value, final state)
21   print $ runState incrementState2 1 -- (1,2) == (return value, final state)
22   print $ runState (mapState bumpVals incrementState) 1 -- (2,4)
23   print $ evalState incrementState 1  -- 1 == return value
24   print $ execState incrementState 1  -- 2 == final state
```

Here we have used two very different looking, yet equivalent, styles for accessing and modifying state monad values. In lines 6-9 we are using the **do** notation. The function **get** in line 7 returns one value: the value wrapped in a state monad. Function **put** in line 8 replaces the wrapped value in the state monad, in this example by incrementing its numeric value. Finally **return** wraps the value in a monad.

I am using the **runState** function defined in lines 20-24 that returns a tuple: the first tuple value is the result of the computation performed by the function passed to **runState** (**incrementState** and **incrementState2** in these examples) and the second tuple value is the final wrapped state.

In lines 12-15 I reimplemented increment state using the *bind* function (>>=). We have seen before that >>= passes the value on its left side to the computation on its right side, that is function calls in lines 13-15:

```
\a -> put (a + 1)
\b -> return a
```

It is a matter of personal taste whether to code using bind or **do**. I almost always use the **do** notation in my own code but I wanted to cover bind both in case you prefer that notation and so you can

also read and understand Haskell code using bind. We continue looking at alternatives to the **do** notation in the next section.

Using Applicative Operators <$> and <*>: Finding Common Words in Files

My goal in this book is to show you a minimal subset of Haskell that is relatively easy to understand and use for coding. However, a big part of using a language is reading other people's code so I do need to introduce a few more constructs that are widely used: applicative operators.

Before we begin I need to introduce you to a new term: **Functor** which is a typeclass that defines only one method **fmap**. **fmap** is used to map a function over an **IO action** and has the type signature:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

fmap can be used to apply a pure function like **(a -> b)** to an **IO a** and return a new **IO b** without unwrapping the original **IO ()**. The following short example (in file *ImPure/FmapExample.hs*) will let you play with this idea:

```
1  module FmapExample where
2
3  fileToWords fileName = do
4    fileText <- readFile fileName
5    return $ words fileText
6
7  main = do
8    words1 <- fileToWords "text1.txt"
9    print $ reverse words1
10   words2 <- fmap reverse $ fileToWords "text1.txt"
11   print words2
```

In lines 8-9 I am unwrapping the result of the **IO [String]** returned by the function **fileToWords** and then applying the pure function **words** to the unwrapped value. Wouldn't it be nice to operate on the words in the file without unwrapping the **[String]** value? You can do this using **fmap** as seen in lines 10-11. Please take a moment to understand what line 10 is doing. Here is line 10:

```
words2 <- fmap reverse $ fileToWords "text1.txt"
```

First we read the words in a file into an **IO [String]** monad:

```
                        fileToWords "text1.txt"
```

Then we apply the pure function **reverse** to the values inside the **IO [String]** monad, creating a new copy:

```
            fmap reverse $ fileToWords "text1.txt"
```

Note that from the type of the **fmap** function, the input monad and output monad can wrap different types. For example, if we applied the function **head** to an **IO [String]** we would get an outut of **IO [Char]**.

Finally we unwrap the [String] value inside the monad and set **words2** to this unwrapped value:

```
  words2 <- fmap reverse $ fileToWords "text1.txt"
```

In summary, the **Functor** typeclass defines one method **fmap** that is useful for operating on data wrapped inside a monad.

We will now implement a small application that finds common words in two text files, implementing the primary function three times, using:

- The **do** notation.
- The >>= bind operator.
- The Applicative operators <$> and <*>

Let's look at the types for these operators:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

We will use both <$> and <*> in the function **commonWords3** in this example and I will explain how these operators work after the following program listing.

This practical example will give you a chance to experiment more with Haskell (you do have a GHCi repl open now, right?). The source file for this example is in the file *ImPure/CommonWords.hs*:

```
1  module CommonWords where
2
3  import Data.Set (fromList, toList, intersection)
4  import Data.Char (toLower)
5
6  fileToWords fileName = do
7    fileText <- readFile fileName
8    return $ (fromList . words) (map toLower fileText)
9
10 commonWords file1 file2 = do
11   words1 <- fileToWords file1
12   words2 <- fileToWords file2
13   return $  toList $ intersection words1 words2
14
15 commonWords2 file1 file2 =
16   fileToWords file1 >>= \f1 ->
17   fileToWords file2 >>= \f2 ->
18   return $  toList $ intersection f1 f2
19
20 commonWords3 file1 file2 =
21   (\f1 f2 -> toList $ intersection f1 f2)
22     <$> fileToWords file1
23     <*> fileToWords file2
24
25 main = do
26   cw <- commonWords "text1.txt" "text2.txt"
27   print cw
28   cw2 <- commonWords "text1.txt" "text2.txt"
29   print cw2
30   cw3 <- commonWords "text1.txt" "text2.txt"
31   print cw3
```

The function **fileToWords** defined in lines 6-8 simply reads a file, as in the last example, maps contents of the file to lower case, uses **words** to convert a **String** to a **[String]** list of individual words, and uses the function **Data.Set.fromList** to create a set from a list of words that in general will have duplicates. We are retuning an **IO (Data.Set.Base.Set String)** value so we can later perform a set intersection operation. In other applications you might want to apply **Data.Set.toList** before returning the value from **fileToWords** so the return type of the function would be **IO [String]**.

The last listing defines three similar functions **commonWords**, **commonWords2**, and **commonWords3**.

commonWords defined in lines 10-13 should hopefully look routine and familiar to you now. We set the local variables with the unwrapped (i.e., extracted from a monad) contents of the unique

words in two files, and then return monad wrapping the intersection of the words in both files.

The function **commonWords2** is really the same as **commonWords** except that it uses the bind >>= operator instead of the **do** notation.

The interesting function in this example is **commonWords3** in lines 20-23 which uses the applicative operators <$> and <*>. Notice the pure function defined inline in line 21: it takes two arguments of type set and returns the set intersection of the arguments. The operator <$> takes a function on the left side and a monad on the right side which contains the wrapped value to be passed as the argument **f1**. <*> supplies the value for the inline function arguments **f2**. To rephrase how lines 21-23 work: we are calling **fileToWords** twice, both times getting a monad. These two wrapped monad values are passed as arguments to the inline function in line 21 and the result of evaluating this inline function is returned as the value of the function **commonWords3**.

I hope that this example has at least provided you with "reading knowledge" of the Applicative operators <$> and <*> and has also given you one more example of replacing the **do** notation with the use of the bind >>= operator.

List Comprehensions Using the do Notation

We saw examples of list comprehensions in the last chapter on pure Haskell programming. We can use **return** to get lists values that are instances of type Monad:

```
*Prelude> :t (return [])
(return []) :: Monad m => m [t]
*Prelude> :t (return [1,2,3])
(return [1,2,3]) :: (Monad m, Num t) => m [t]
*Prelude> :t (return ["the","tree"])
(return ["the","tree"]) :: Monad m => m [[Char]]
```

We can get list comprehension behavior from the **do** notation (here I am using the GHCi repl **:{** and **:}** commands to enter multiple line examples):

```
1  *Main> :{
2  *Main| do num <- [1..3]
3  *Main|    animal <- ["parrot", "ant", "dolphin"]
4  *Main|    return (num, animal)
5  *Main| :}
6  [(1,"parrot"),(1,"ant"),(1,"dolphin"),
7   (2,"parrot"),(2,"ant"),(2,"dolphin"),
8   (3,"parrot"),(3,"ant"),(3,"dolphin")]
```

I won't use this notation further but you now will recognize this pattern if you read it in other people's code.

Dealing With Time

In the example in this section we will see how to time a block of code (using two different methods) and how to set a timeout for code that runs in an **IO ()**.

The first way we time a block of code uses **getPOSIXTime** and can be used to time pure or impure code. The second method using **timeIt** takes an **IO ()** as an argument; in the following example I wrapped pure code in a **print** function call which returns an **IO ()** as its value. The last example in the file *TimerTest.hs* shows how to run impure code wrapped in a timeout.

```
module Main where

import Data.Time.Clock.POSIX -- for getPOSIXTime
import System.TimeIt         -- for timeIt
import System.Timeout        -- for timeout

anyCalculationWillDo n =  -- a function that can take a while to run
  take n $ sieve [2..]
           where
             sieve (x:xs) =
               x:sieve [y | y <- xs, rem y x > 0]

main = do
  startingTime <- getPOSIXTime
  print startingTime
  print $ last $ take 20000001 [0..]
  endingTime <- getPOSIXTime
  print endingTime
  print (endingTime - startingTime)
  timeIt $ print $ last $ anyCalculationWillDo 2000

  let somePrimes = anyCalculationWillDo 3333 in
    timeIt $ print $ last somePrimes

  -- 100000 microseconds timeout tests:
  timeout 100000 $ print "simple print **do** expression did not timeout"
  timeout 100000 $ print $ last $ anyCalculationWillDo 4
  timeout 100000 $ print $ last $ anyCalculationWillDo 40
  timeout 100000 $ print $ last $ anyCalculationWillDo 400
  timeout 100000 $ print $ last $ anyCalculationWillDo 4000
  timeout 100000 $ print $ last $ anyCalculationWillDo 40000
  print $ anyCalculationWillDo 5
```

I wanted a function that takes a while to run so for **anyCalculationWillDo** (lines 7 to 11) I implemented an inefficient prime number generator.

When running this example on my laptop, the last two timeout calls (lines 26 and 31) are terminated for taking more than 100000 microseconds to execute.

The last line 32 of code prints out the first 5 prime numbers greater than 1 so you can see the results of calling the time wasting test function **anyCalculationWillDo**.

```
1  $ stack build --exec TimerTest
2  1473610528.2177s
3  20000000
4  1473610530.218574s
5  2.000874s
6  17389
7  CPU time:   0.14s
8  30911
9  CPU time:   0.25s
10 "simple print **do** expression did not timeout"
11 7
12 173
13 2741
14 [2,3,5,7,11]
```

The **timeout** function is useful for setting a maximum time that you are willing to wait for a calculation to complete. I mostly use **timeout** for timing out operations fetching data from the web.

Wrap Up

I tried to give you a general fast-start in this chapter for using monads and in general writing impure Haskell code. This chapter should be sufficient for you to be able to understand and experiment with the examples in the rest of this book.

This is the end of the first section. We will now look at a variety of application examples using the Haskell language.

While I expect you to have worked through the previous chapters in order, for the rest of the book you can skip around and read the material in any order that you wish.

Section 2 - Cookbook

Now that you have worked through the pure and impure Haskell coding tutorials in the first two chapters we will look at a "cookbook" of techniques and sample applications to solve some common programming tasks as well as implement a program to play the card game Blackjack.

I expect you, dear reader, to have studied and absorbed the tutorial material on pure and impure Haskell programming in the first two chapters. If you are new to Haskell, or don't have much experience yet, carefully working through these tutorial chapters is a requirement for understanding the material in the rest of this book.

This section contains the following "recipe" applications:

- Textprocessing CSV Files
- Textprocessing JSON Files
- Using sqlite and Postgres databases
- Natural Language Processing (NLP) interface to relational databases
- REST Server Providing JSON Data
- REST Client
- Accessing and Using Linked Data
- Querying Semantic Web RDF Data Sources
- Annotating English text with Wikipedia/DBPedia URIs for entities in the original text. Entities can be people, places, organizations, etc.
- Play the Blackjack card game
- Machine Learning
- Probabilistic Graph Models

Text Processing

In my work in data science and machine learning, processing text is a core activity. I am a practitioner, not a research scientist, and in a practical sense, I spend a fair amount of time collecting data (e.g., web scraping and using semantic web/linked data sources), cleaning it, and converting it to different formats.

We will cover three useful techniques: parsing and using CSV (comma separated values) spreadsheet files, parsing and using JSON data, and cleaning up natural language text that contains noise characters.

CSV Spreadsheet Files

The comma separated values (CSV) format is a plain text format that all spreadsheet applications support. The following example illustrates two techniques that we haven't covered yet:

- Extracting values from the **Either** type.
- Using destructuring to concisely extract parts of a list.

The **Either** type *Either a b* contains either a *Left a* or a *Right b* value and is usually used to return an error in **Left** or a value in **Right**. We will using the **Data.Either.Unwrap** module to unwrap the **Right** part of a call to the **Text.CSV.parseCSVFromFile** function that reads a CSV file and returns a **Left** error or the data in the spreadsheet in a list as the **Right** value.

The destructuring trick in line 14 in the following listing lets us separate the head and rest of a list in one operation; for example:

```
*TestCSV> let z = [1,2,3,4,5]
*TestCSV> z
[1,2,3,4,5]
*TestCSV> let x:xs = z
*TestCSV> x
1
*TestCSV> xs
[2,3,4,5]
```

Here is how to read a CSV file:

```
module TestCSV where

import Text.CSV (parseCSVFromFile, CSV)
import Data.Either.Unwrap (fromRight)

readCsvFile :: FilePath -> CSV
readCsvFile fname = do
  c <- parseCSVFromFile fname
  return $ fromRight c

main = do
  c <- readCsvFile "test.csv"
  print  c  -- includes header and data rows
  print $ map head c  -- print header
  let header:rows = c -- destructure
  print header
  print rows
```

Function **readCsvFile** reads from a file and returns a **CSV**. What is a **CSV** type? You could search the web for documentation, but dear reader, if you have worked this far learning Haskell, by now you know to rely on the GHCi repl:

```
*TestCSV> :i CSV
type CSV = [Text.CSV.Record]          -- Defined in ‘Text.CSV’
*TestCSV> :i Text.CSV.Record
type Text.CSV.Record = [Text.CSV.Field]          -- Defined in ‘Text.CSV’
*TestCSV> :i Text.CSV.Field
type Text.CSV.Field = String          -- Defined in ‘Text.CSV’
```

So, a **CSV** is a list of records (rows in the spreadsheet file), each record is a list of fields (i.e., a string value).

The output when reading the CVS file *test.csv* is:

```
Prelude> :l TestCSV
[1 of 1] Compiling TestCSV          ( TestCSV.hs, interpreted )
Ok, modules loaded: TestCSV.
*TestCSV> main
[["name"," email"," age"],["John Smith"," jsmith@acmetools.com"," 41"],["June Jo\
nes"," jj@acmetools.com"," 38"]]
["name","John Smith","June Jones"]
["name"," email"," age"]
[["John Smith"," jsmith@acmetools.com"," 41"],["June Jones"," jj@acmetools.com",\
" 38"]]
```

JSON Data

JSON is the native data format for the Javascript language and JSON has become a popular serialization format for exchanging data between programs on a network. In this section I will demonstrate serializing a Haskell type to a string with JSON encoding and then perform the opposite operation of deserializing a string containing JSON encoded data back to an object.

The first example uses the module **Text.JSON.Generic** (from the *json* library) and the second example uses module **Data.Aeson** (from the *aeson* library).

In the first example, we set the language type to include DeriveDataTypeable so a new type definition can simply derive *Typeable* which allows the compiler to generate appropriate **encodeJSON** and **decodeJSON** functions for the type **Person** we define in the example:

```
1  {-# LANGUAGE DeriveDataTypeable #-}
2
3  module TestTextJSON where
4
5  import Text.JSON.Generic
6
7  data Person = Person {name::String, email::String}
8                       deriving (Show, Data, Typeable)
9
10 main = do
11   let a = encodeJSON $ Person "Sam" "sam@a.com"
12   print a
13   let d = (decodeJSON a :: Person)
14   print d
15   print $ name d
16   print $ email d
```

Notice that in line 14 that I specified the expected type in the **decodeJSON** call. This is not strictly required, the Haskell GHC compiler knows what to do in this case. I specified the type for code

readability. The Haskell compiler wrote the **name** and **email** functions for me and I use these functions in lines 16 and 17 to extract these fields. Here is the output from running this example:

```
1  Prelude> :l TestTextJSON.hs
2  [1 of 1] Compiling TestTextJSON     ( TestTextJSON.hs, interpreted )
3  Ok, modules loaded: TestTextJSON.
4  *TestTextJSON> main
5  "{\"name\":\"Sam\",\"email\":\"sam@a.com\"}"
6  Person {name = "Sam", email = "sam@a.com"}
7  "Sam"
8  "sam@a.com"
```

The next example uses the *Aeson* library and is similar to this example.

Using *Aeson*, we set a language type *DeriveGeneric* and in this case have the **Person** class derive **Generic**. The School of Haskell has an excellent Aeson tutorial[21] that shows a trick I use in this example: letting the compiler generate required functions for types **FromJSON** and **ToJSON** as seen in lines 12-13.

```
1  {-# LANGUAGE DeriveGeneric #-}
2
3  module TestJSON where
4
5  import Data.Aeson
6  import GHC.Generics
7  import Data.Maybe
8
9  data Person = Person {name::String, email::String } deriving (Show, Generic)
10
11 -- nice trick from School Of Haskell tutorial on Aeson:
12 instance FromJSON Person  -- DeriveGeneric language setting allows
13 instance ToJSON Person    -- automatic generation of instance of
14                          -- types deriving Generic.
15
16 main = do
17   let a = encode $ Person "Sam" "sam@a.com"
18   print a
19   let (Just d) = (decode a :: Maybe Person)
20   print d
21   print $ name d
22   print $ email d
```

[21] https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json

I use a short cut in line 19, assuming that the **Maybe** object returned from **decode** (which the compiler wrote automatically for the type **FromJSON**) contains a **Just** value instead of an empty **Nothing** value. So in line 19 I directly unwrap the **Just** value.

Here is the output from running this example:

```
1  Prelude> :l TestAESON.hs
2  [1 of 1] Compiling TestJSON         ( TestAESON.hs, interpreted )
3  Ok, modules loaded: TestJSON.
4  *TestJSON> main
5  "{\"email\":\"sam@a.com\",\"name\":\"Sam\"}"
6  Person {name = "Sam", email = "sam@a.com"}
7  "Sam"
8  "sam@a.com"
```

Line 5 shows the result of printing the JSON encoded string value created by the call to **encode** in line 17 of the last code example. Line 6 shows the decoded value of type **Person**, and lines 7 and 8 show the inner wrapped values in the **Person** data.

Cleaning Natural Language Text

I spend a lot of time working with text data because I have worked on NLP (natural language processing) projects for over 25 years. We will jump into some interesting NLP applications in the next chapter. I will finish this chapter with strategies for cleaning up text which is often a precursor to performing NLP.

You might be asking why we would need to clean up text. Here are a few common use cases:

- Text fetched from the web frequently contains garbage characters.
- Some types of punctuation need to be removed.
- Stop words (e.g., the, a, but, etc.) need to be removed.
- Special unicode characters are not desired.
- Sometimes we want white space around punctuation to make tokenizing text easier.

Notice the **module** statement on line 1 of the following listing: I am exporting functions **cleanText** and **removeStopWords** so they will be visible and available for use by any other modules that import this module. In line 6 we import **intercalate** which constructs a string from a space character and an [String] (i.e., a list of strings); here is an example where instead of adding a space character between the strings joined together, I add "*" characters:

```
*CleanText> intercalate "*" ["the", "black", "cat"]
"the*black*cat"
```

The function **cleanText** removes garbage characters and makes sure that any punctuation characters are surrounded by white space (this makes it easier, for example, to determine sentence boundaries). Function **removeStopWords** removes common words like "a", "the", etc. from text.

```
module CleanText (cleanText, removeStopWords)  where

import Data.List.Split (splitOn)
import Data.List (intercalate)
import Data.Char as C
import Data.List.Utils (replace)

noiseCharacters = ['[', ']', '{', '}', '\n', '\t', '&', '^',
                   '@', '%', '$', '#', ',']

substituteNoiseCharacters :: [Char] -> [Char]
substituteNoiseCharacters =
  map (\x -> if elem x noiseCharacters then ' ' else x)

cleanText s =
  intercalate
   " " $
   filter
     (\x -> length x > 0) $
     splitOn " " $ substituteNoiseCharacters $
       (replace "." " . "
        (replace "," " , "
         (replace ";" " ; " s)))

stopWords = ["a", "the", "that", "of", "an"]

toLower' :: [Char] -> [Char]
toLower' s = map (\x -> if isLower x then x else (C.toLower x)) s

removeStopWords :: String -> [Char]
removeStopWords s =
  intercalate
     " " $
    filter
      (\x -> notElem (toLower' x) stopWords) $
```

```
      words s

main = do
  let ct = cleanText "The[]@] cat, and all dog, escaped&^. They were caught."
  print ct
  let nn = removeStopWords ct
  print nn
```

This example should be extended with additional noise characters and stop words, depending on your application.

Here is the output from this example:

```
*TestCleanText> :l CleanText.hs
[1 of 1] Compiling TestCleanText    ( CleanText.hs, interpreted )
Ok, modules loaded: TestCleanText.
*TestCleanText> main
"The cat and all dog escaped . They were caught ."
"cat dog escaped . They were caught ."
```

We will continue working with text in the next chapter.

Natural Language Processing Tools

The tools developed in this chapter are modules you can reuse in your programs. We will develop a command line program that reads a line of text from STDIN and writes sematic information as output to STDOUT. I have used this in a Ruby program by piping input text data to a forked process and reading the output which is a semantic representation of the input text.

A few of the data files I provide in this example are fairly large. As an example the file *PeopleDbPedia.hs* which builds a map from people's names to the Wikipedia/DBPedia URI for information about them, is 2.5 megabytes in size. The first time you run *stack build* in the project directory it will take a while, so you might want to start building the project in the directory *NlpTool* and let it run while you read this chapter.

Here are three examples using the NlpTool command line application developed in this chapter:

```
Enter text (all on one line)
Canada and England signed a trade deal.
category:        economics
summary:        Canada and England signed a trade deal.
countries:         [["Canada","<http://dbpedia.org/resource/Canada>"],
             ["England","<http://dbpedia.org/resource/England>"]]
Enter text (all on one line)
President George W Bush asked Congress for permission to invade Iraq.
category:        news_war
summary:        President George W Bush asked Congress for permission to invade Iraq.
people:        [["George W Bush","<http://dbpedia.org/resource/George_W._Bush>"]]
countries:         [["Iraq",""]]
Enter text (all on one line)
The British government is facing criticism from business groups over statements \
suggesting the U.K. is heading for a hard divorce from the European Union â€” an\
d pressure from lawmakers who want Parliament to have a vote on the proposed exi\
t terms. The government's repeated emphasis on controlling immigration sent out \
"signs that the door is being closed, to an extent, on the open economy, that ha\
s helped fuel investment," the head of employers' group the Confederation of Bri\
tish Industry, Carolyn Fairbairn, said in comments published Monday. Prime Minis\
ter Theresa May said last week that Britain would seek to retain a close relatio\
nship with the 28-nation bloc, with continued free trade in goods and services. \
But she said the U.K. wouldn't cede control over immigration, a conflict with th\
e EU's principle of free movement among member states.
category:        economics
```

```
summary:        Prime Minister Theresa May said last week that Britain would seek to re\
tain a close relationship with the 28-nation bloc, with continued free trade in \
goods and services.
```

credit: news text from abcnews.com

Resolve Entities in Text to DBPedia URIs

The code for this application is in the directory *NlpTool*.

The software and data in this chapter can be used under the terms of either the GPL version 3 license or the Apache 2 license.

There are several automatically generated Haskell formatted data files that I created using Ruby scripts operating the Wikipedia data. For the purposes of this book I include these data-specific files for your use and enjoyment but we won't spend much time discussing them. These files are:

- BroadcastNetworkNamesDbPedia.hs
- CityNamesDbpedia.hs
- CompanyNamesDbpedia.hs
- CountryNamesDbpedia.hs
- PeopleDbPedia.hs
- PoliticalPartyNamesDbPedia.hs
- TradeUnionNamesDbPedia.hs
- UniversityNamesDbPedia.hs

As an example, let's look at a small sample of data in *PeopleDbPedia.hs*:

```
1  module PeopleDbPedia (peopleMap) where
2
3  import qualified Data.Map as M
4
5  peopleMap = M.fromList [
6    ("Aaron Sorkin", "<http://dbpedia.org/resource/Aaron_Sorkin>"),
7    ("Bill Clinton", "<http://dbpedia.org/resource/Bill_Clinton>"),
8    ("George W Bush", "<http://dbpedia.org/resource/George_W_Bush>"),
```

There are 35,146 names in the file *PeopleDbPedia.hs*. I have built for eight different types of entity names: Haskell maps that take entity names (String) and maps the entity names into relevant DBPedia URIs. Simple in principle, but a lot of work preparing the data. As I mentioned, we will use these data-specific files to resolve entity references in text.

The next listing shows the file *Entities.hs*. In lines 5-7 I import the entity mapping files I just described. In this example and later code I make heavy use of the **Data.Map** and **Data.Set** modules in the *collections* library (see the NlpTools.cabal file).

The operator **`isSubsetOf`** defined in line 39 tests to see if a value is contained in a collection. The built-in function **all** applies a function or operator to all elements in a collection and returns a true value if the function or operator returns true applied to each element in the collection.

The local utility function **namesHelper** defined in lines 41-53 is simpler than it looks. The function **filter** in line 42 applies the inline function in lines 43-45 (this function returns true fo Maybe values that contain data) to a second list defined in lines 48-55. This second list is calculated by mapping an inline function over the input argument **ngrams**. The inline function looks up an ngram in a DBPedia map (passed as the second function argument) and returns the lookup value if it is not empty and if it is empty looks up the same ngram in a word map (last argument to this function).

The utility function **namesHelper** is then used to define functions to recognize company names, country names, people names, city names, broadcast network names, political party names, trade union names, and university names:

```
1  -- Copyright 2014 by Mark Watson. All rights reserved. The software and data in \
2  this project can be used under the terms of either the GPL version 3 license or \
3  the Apache 2 license.
4
5  module Entities (companyNames, peopleNames,
6                   countryNames, cityNames, broadcastNetworkNames,
7                   politicalPartyNames, tradeUnionNames, universityNames) where
8
9  import qualified Data.Map as M
10 import qualified Data.Set as S
11 import Data.Char (toLower)
12 import Data.List (sort, intersect, intersperse)
13 import Data.Set (empty)
14 import Data.Maybe (isJust)
15
16 import Utils (splitWords, bigram, bigram_s, splitWordsKeepCase,
17               trigram, trigram_s, removeDuplicates)
18
19 import FirstNames (firstNames)
20 import LastNames (lastNames)
21 import NamePrefixes (namePrefixes)
22
23 import PeopleDbPedia (peopleMap)
24
25 import CountryNamesDbpedia (countryMap)
```

```
26  import CountryNames (countryNamesOneWord, countryNamesTwoWords, countryNamesThre\
27  eWords)
28
29  import CompanyNamesDbpedia (companyMap)
30  import CompanyNames (companyNamesOneWord, companyNamesTwoWords, companyNamesThre\
31  eWords)
32  import CityNamesDbpedia (cityMap)
33
34  import BroadcastNetworkNamesDbPedia (broadcastNetworkMap)
35  import PoliticalPartyNamesDbPedia (politicalPartyMap)
36  import TradeUnionNamesDbPedia (tradeUnionMap)
37  import UniversityNamesDbPedia (universityMap)
38
39  xs `isSubsetOf` ys = all (`elem` ys) xs
40
41  namesHelper ngrams dbPediaMap wordMap =
42    filter
43      (\x -> case x of
44           (_, Just x) -> True
45           _ -> False) $
46      map (\ngram -> (ngram,
47                  let v = M.lookup ngram dbPediaMap in
48                  if isJust v
49                     then return (ngram, v)
50                     else if S.member ngram wordMap
51                             then Just (ngram, Just "")
52                             else Nothing))
53          ngrams
54
55  helperNames1W = namesHelper
56
57  helperNames2W wrds = namesHelper (bigram_s wrds)
58
59  helperNames3W wrds =  namesHelper (trigram_s wrds)
60
61  companyNames wrds =
62    let cns = removeDuplicates $ sort $
63                helperNames1W wrds companyMap companyNamesOneWord ++
64                helperNames2W wrds companyMap companyNamesTwoWords ++
65                helperNames3W wrds companyMap companyNamesThreeWords in
66    map (\(s, Just (a,Just b)) -> (a,b)) cns
67
```

```
68  countryNames wrds =
69    let cns = removeDuplicates $ sort $
70                helperNames1W wrds countryMap countryNamesOneWord ++
71                helperNames2W wrds countryMap countryNamesTwoWords ++
72                helperNames3W wrds countryMap countryNamesThreeWords in
73    map (\(s, Just (a,Just b)) -> (a,b)) cns
74
75  peopleNames wrds =
76    let cns = removeDuplicates $ sort $
77                helperNames1W wrds peopleMap Data.Set.empty ++
78                helperNames2W wrds peopleMap Data.Set.empty ++
79                helperNames3W wrds peopleMap Data.Set.empty in
80    map (\(s, Just (a,Just b)) -> (a,b)) cns
81
82  cityNames wrds =
83    let cns = removeDuplicates $ sort $
84                helperNames1W wrds cityMap Data.Set.empty ++
85                helperNames2W wrds cityMap Data.Set.empty ++
86                helperNames3W wrds cityMap Data.Set.empty in
87    map (\(s, Just (a,Just b)) -> (a,b)) cns
88
89  broadcastNetworkNames wrds =
90    let cns = removeDuplicates $ sort $
91                helperNames1W wrds broadcastNetworkMap Data.Set.empty ++
92                helperNames2W wrds broadcastNetworkMap Data.Set.empty ++
93                helperNames3W wrds broadcastNetworkMap Data.Set.empty in
94    map (\(s, Just (a,Just b)) -> (a,b)) cns
95
96  politicalPartyNames wrds =
97    let cns = removeDuplicates $ sort $
98                helperNames1W wrds politicalPartyMap Data.Set.empty ++
99                helperNames2W wrds politicalPartyMap Data.Set.empty ++
100               helperNames3W wrds politicalPartyMap Data.Set.empty in
101   map (\(s, Just (a,Just b)) -> (a,b)) cns
102
103 tradeUnionNames wrds =
104   let cns = removeDuplicates $ sort $
105               helperNames1W wrds tradeUnionMap Data.Set.empty ++
106               helperNames2W wrds tradeUnionMap Data.Set.empty ++
107               helperNames3W wrds tradeUnionMap Data.Set.empty in
108   map (\(s, Just (a,Just b)) -> (a,b)) cns
109
```

```
universityNames wrds =
  let cns = removeDuplicates $ sort $
            helperNames1W wrds universityMap Data.Set.empty ++
            helperNames2W wrds universityMap Data.Set.empty ++
            helperNames3W wrds universityMap Data.Set.empty in
  map (\(s, Just (a,Just b)) -> (a,b)) cns


main = do
    let s = "As read in the San Francisco Chronicle, the company is owned by Joh\
n Smith, Bill Clinton, Betty Sanders, and Dr. Ben Jones. Ben Jones and Mr. John \
Smith are childhood friends who grew up in Brazil, Canada, Buenos Aires, and the\
 British Virgin Islands. Apple Computer relased a new version of OS X yesterday.\
 Brazil Brazil Brazil. John Smith bought stock in ConocoPhillips, Heinz, Hasbro,\
 and General Motors, Fox Sports Radio. I listen to B J Cole. Awami National Part\
y is a political party. ALAEA is a trade union. She went to Brandeis University.\
"
    --print $ humanNames s
    print $ peopleNames $ splitWordsKeepCase s
    print $ countryNames $ splitWordsKeepCase s
    print $ companyNames $ splitWordsKeepCase s
    print $ cityNames $ splitWordsKeepCase s
    print $ broadcastNetworkNames $ splitWordsKeepCase s
    print $ politicalPartyNames $ splitWordsKeepCase s
    print $ tradeUnionNames $ splitWordsKeepCase s
    print $ universityNames $ splitWordsKeepCase s
```

The following output is generated by running the test **main** function defined at the bottom of the file *Entities.hs*:

```
$ stack ghci
UniversityNamesDbPedia Utils> :l Entities.hs
Ok, modules loaded: Utils, UniversityNamesDbPedia, TradeUnionNamesDbPedia,
PoliticalPartyNamesDbPedia, PeopleDbPedia, NamePrefixes, LastNames,
FirstNames, Entities, CountryNamesDbpedia, CountryNames, CompanyNamesDbpedia,
CompanyNames, CityNamesDbpedia, BroadcastNetworkNamesDbPedia.
*Entities> main
[("Bill Clinton","<http://dbpedia.org/resource/Bill_Clinton>")]
[("Brazil","<http://dbpedia.org/resource/Brazil>"),
 ("British Virgin Islands",""),
 ("Buenos Aires",""),
 ("Canada","<http://dbpedia.org/resource/Canada>"),("Virgin Islands","")]
```

```
[("Apple","<http://dbpedia.org/resource/Apple>"),
("Apple Computer","<http://dbpedia.org/resource/Apple_Inc.>"),
("ConocoPhillips","<http://dbpedia.org/resource/ConocoPhillips>"),
("General Motors","<http://dbpedia.org/resource/General_Motors>"),
("Hasbro","<http://dbpedia.org/resource/Hasbro>"),
("Heinz","<http://dbpedia.org/resource/Heinz>"),
("San Francisco Chronicle","<http://dbpedia.org/resource/San_Francisco_Chronicle\
>")]
[("San Francisco","<http://dbpedia.org/resource/San_Francisco>")]
[("Fox Sports Radio","<http://dbpedia.org/resource/Fox_Sports_Radio>")]
[("Awami National Party","<http://dbpedia.org/resource/Awami_National_Party>"),
 ("National Party","<http://dbpedia.org/resource/National_Party_(Syria)>")]
[("ALAEA","<http://dbpedia.org/resource/Australian_Licenced_Aircraft_Engineers_A\
ssociation>")]
[]
*Entities>
```

Note that in lines 10 and 11, the British Virgin Islands and Buenos Aires are recognized as locations but the example program was unable to resolve them to Wikipedia/DBPedia URIs.

Bag of Words Classification Model

The file *Categorize.hs* contains a simple bag of words classification model. To prepare the classification models, I collected a large set of labelled text. Labels were "chemistry", "computers", etc. I ranked words based on how often they appeared in training texts for a classification category, normalized by how often they appeared in all training texts. This example uses two auto-generated and data-specific Haskell files, one for single words and the other for two adjacent word pairs:

- Category1Gram.hs
- Category2Gram.hs

In NLP work, single words are sometimes called 1grams and two word adjacent pairs are referred to as 2grams. Here is a small amount of data from *Category1Gram.hs*:

```
module Category1Gram (**onegrams**) where

import qualified Data.Map as M

chemistry = M.fromList [("chemical", 1.15), ("atoms", 6.95),
                        ("reaction", 6.7), ("energy", 6.05),
                          ... ]
computers = M.fromList [("software", 4.6), ("network", 4.65),
                        ("linux", 3.6), ("device", 3.55), ("computers", 3.05),
                        ("storage", 2.7), ("disk", 2.3),
                          ... ]
etc.
```

Here is a small amount of data from *Category2Gram.hs*:

```
module Category2Gram (**twograms**) where

import qualified Data.Map as M

chemistry = M.fromList [("chemical reaction", 1.55),
                        ("atoms molecules", 0.6),
                        ("periodic table", 0.5),
                        ("chemical reactions", 0.5),
                        ("carbon atom", 0.5),
                         ... ]
computers = M.fromList [("computer system", 0.9),
                        ("operating system", 0.75),
                        ("random memory", 0.65),
                        ("computer science", 0.65),
                        ("computer program", 0.6),
                         ... ]
etc.
```

It is very common to use term frequencies for single words for classification models. One problem with using single words is that the evidence that any word gives for a classification is independent of the surrounding words in text being evaluated. By also using word pairs (two word combinations are often called 2grams or two-grams) we pick up patterns like "not good" giving evidence for negative sentiment even with the word "good" in text being evaluated. For my own work, I have a huge corpus of 1gram, 2gram, 3gram, and 4gram data sets. For the purposes of the following example program, I am only using 1gram and 2gram data.

The following listing shows the file *Categorize.hs*. Before looking at the entire example, let's focus in on some of the functions I have defined for using the word frequency data to categorized text.

```
*Categorize> :t stemWordsInString
stemWordsInString :: String -> [Char]
*Categorize> stemWordsInString "Banking industry is sometimes known for fraud."
"bank industri is sometim known for fraud"
```

stemScoredWordList is used to create a 1gram to word relevance score for each category. The keys are word stems.

```
*Categorize> stemScoredWordList onegrams
[("chemistri",fromList [("acid",1.15),("acids",0.8),("alcohol",0.95),("atom",4.4\
5)
```

Notice that "chemistri" is the stemmed version of "chemistry", "bank" for "banks", etc. **stem2** is a 2gram frequency score by category mapping where the keys are word stems:

```
*Categorize> stem2
[("chemistry",fromList [("atom molecul",0.6),("carbon atom",0.5),("carbon carbon\
",0.5),
```

stem1 is like **stem2**, but for stemmed 1grams, not 2grams:

```
*Categorize> stem1
[("chemistry",fromList [("acid",0.8),("chang",1.05),("charg",0.95),("chemic",1.1\
5),("chemistri",1.45),
```

score is called with a list or words and a word value mapping. Here is an example:

```
*Categorize> :t score
score
  :: (Enum t, Fractional a, Num t, Ord a, Ord k) =>
     [k] -> [(t1, M.Map k a)] -> [(t, a)]
*Categorize> score ["atom", "molecule"] onegrams
[(0,8.2),(25,2.4)]
```

This output is more than a little opaque. The pair (0, 8.2) means that the input words ["atom", "molecule"] have a score of 8.2 for category indexed at 0 and the pair (25,2.4) means that the input words have a score of 2.4 for the category at index 25. The category at index 0 is chemistry and the category at index 25 is physics as we can see by using the higher level function **bestCategories1** that caluculates categories for a word sequence using 1gram word data:

```
*Categorize> :t bestCategories1
bestCategories1 :: [[Char]] -> [([Char], Double)]
*Categorize> bestCategories1 ["atom", "molecule"]
[("chemistry",8.2),("physics",2.4)]
```

The top level function **bestCategories** uses 1gram data. Here is an example for using it:

```
*Categorize> splitWords "The chemist made a periodic table and explained a chemi\
cal reaction"
["the","chemist","made","a","periodic","table","and","explained","a","chemical",\
"reaction"]
*Categorize> bestCategories1 $ splitWords "The chemist made a periodic table and\
 explained a chemical reaction"
[("chemistry",11.25),("health_nutrition",1.2)]
```

Notice that these words were also classified as category "health_nutrition" but with a low score of 1.2. The score for "chemistry" is almost an order of magnitude larger. **bestCategories** sorts return values in "best first" order.

splitWords is used to split a string into word tokens before calling **bestCategories**.

Here is the entire example in file *Categorize.hs*:

```
1  module Categorize (bestCategories, splitWords, bigram) where
2
3  import qualified Data.Map as M
4  import Data.List (sortBy)
5
6  import Category1Gram (onegrams)
7  import Category2Gram (twograms)
8
9  import Sentence (segment)
10
11 import Stemmer (stem)
12
13 import Utils (splitWords, bigram, bigram_s)
14
15 catnames1 = map fst onegrams
16 catnames2 = map fst twograms
17
18 stemWordsInString s = init $ concatMap ((++ " ") . stem) (splitWords s)
19
20 stemScoredWordList = map (\(str,score) -> (stemWordsInString str, score))
```

```
21
22 stem2 = map (\(category, swl) ->
23               (category, M.fromList (stemScoredWordList (M.toList swl))))
24                         twograms
25
26 stem1 = map (\(category, swl) ->
27               (category, M.fromList (stemScoredWordList (M.toList swl))))
28                     onegrams
29
30 scoreCat wrds amap =
31   sum $ map (\x ->  M.findWithDefault 0.0 x amap) wrds
32
33 score wrds amap =
34  filter (\(a, b) -> b > 0.9) $ zip [0..] $ map (\(s, m) -> scoreCat wrds m) amap
35
36 cmpScore (a1, b1) (a2, b2) = compare b2 b1
37
38 bestCategoriesHelper wrds ngramMap categoryNames=
39   let tg = bigram_s wrds in
40     map (first (categoryNames !!)) $ sortBy cmpScore $ score wrds ngramMap
41
42 bestCategories1 wrds =
43   take 3 $ bestCategoriesHelper wrds onegrams catnames1
44
45 bestCategories2 wrds =
46   take 3 $ bestCategoriesHelper (bigram_s wrds) twograms catnames2
47
48 bestCategories1stem wrds =
49   take 3 $ bestCategoriesHelper wrds stem1 catnames1
50
51 bestCategories2stem wrds =
52   take 3 $ bestCategoriesHelper (bigram_s wrds) stem2 catnames2
53
54 bestCategories :: [String] -> [(String, Double)]
55 bestCategories wrds =
56   let sum1 = M.unionWith (+) (M.fromList $ bestCategories1 wrds) ( M.fromList $ \
57 bestCategories2 wrds)
58       sum2 = M.unionWith (+) (M.fromList $ bestCategories1stem wrds) ( M.fromLis\
59 t $ bestCategories2stem wrds)
60   in sortBy cmpScore $ M.toList $ M.unionWith (+) sum1 sum2
61
62 main = do
```

```
63      let s = "The sport of hocky is about 100 years old by ahdi dates. American F\
64  ootball is a newer sport. Programming is fun. Congress passed a new budget that \
65  might help the economy. The frontier initially was a value path. The ai research\
66   of john mccarthy."
67      print $ bestCategories1 (splitWords s)
68      print $ bestCategories1stem (splitWords s)
69      print $ score (splitWords s) onegrams
70      print $ score (bigram_s (splitWords s)) twograms
71      print $ bestCategories2 (splitWords s)
72      print $ bestCategories2stem (splitWords s)
73      print $ bestCategories (splitWords s)
```

Here is the output:

```
1  $ stack ghci
2  :l Categorize.hs
3  *Categorize> main
4  [("computers_ai",17.900000000000002),("sports",9.75),("computers_ai_search",6.2)]
5  [("computers_ai",18.700000000000003),("computers_ai_search",8.1),("computers_ai_\
6  learning",5.7)]
7  [(2,17.900000000000002),(3,1.75),(4,5.05),(6,6.2),(9,1.1),(10,1.2),(21,2.7),(26,\
8  1.1),(28,1.6),(32,9.75)]
9  [(2,2.55),(6,1.0),(32,2.2)]
10 [("computers_ai",2.55),("sports",2.2),("computers_ai_search",1.0)]
11 [("computers_ai",1.6)]
12 [("computers_ai",40.75000000000001),("computers_ai_search",15.3),("sports",11.95\
13 ),("computers_ai_learning",5.7)]
```

Given that the variable **s** contains some test text, line 4 of this output was generated by evaluating **bestCategories1 (splitWords s)**, lines 5-6 by evaluating **bestCategories1stem (splitWords s)**, lines 7-8 from **score (splitWords s) onegrams**, line 9 from **core (bigram_s (splitWords s)) twograms**, line 10 from **bestCategories2 (splitWords s)**, line 11 from **bestCategories2stem (splitWords s)**, and lines 12-13 from **bestCategories (splitWords s)**.

I called all of the utility fucntions in function **main** to demonstrate what they do but in practice I just call function **bestCategories** in my applications.

Text Summarization

This application uses both the *Categorize.hs* code and the 1gram data from the last section. The algorithm I devised for this example is based on a simple idea: we categorize text and keep track of which words provide the strongest evidence for the highest ranked categories. We then return a few sentences from the original text that contain the largest numbers of these important words.

```
module Summarize (summarize, summarizeS) where

import qualified Data.Map as M
import Data.List.Utils (replace)
import Data.Maybe (fromMaybe)

import Categorize (bestCategories)
import Sentence (segment)
import Utils (splitWords, bigram_s, cleanText)

import Category1Gram (onegrams)
import Category2Gram (twograms)

scoreSentenceHelper words scoreMap = -- just use 1grams for now
  sum $ map (\word ->  M.findWithDefault 0.0 word scoreMap) words

safeLookup key alist =
  fromMaybe 0 $ lookup key alist

scoreSentenceByBestCategories words catDataMaps bestCategories =
  map (\(category, aMap) ->
        (category, safeLookup category bestCategories *
                   scoreSentenceHelper words aMap)) catDataMaps

scoreForSentence words catDataMaps bestCategories =
  sum $ map snd $ scoreSentenceByBestCategories words catDataMaps bestCategories

summarize s =
  let words = splitWords $ cleanText s
      bestCats = bestCategories words
      sentences = segment s
      result1grams = map (\sentence ->
                           (sentence,
                            scoreForSentence (splitWords sentence)
                                             onegrams bestCats))
                         sentences
      result2grams = map (\sentence ->
                           (sentence,
                            scoreForSentence (bigram_s (splitWords sentence))
                                             twograms bestCats))
                         sentences
      mergedResults = M.toList $ M.unionWith (+)
```

```
                    (M.fromList result1grams) (M.fromList result1grams)
      c400 = filter (\(sentence, score) -> score > 400) mergedResults
      c300 = filter (\(sentence, score) -> score > 300) mergedResults
      c200 = filter (\(sentence, score) -> score > 200) mergedResults
      c100 = filter (\(sentence, score) -> score > 100) mergedResults
      c000 = mergedResults in
  if not (null c400) then c400 else if not (null c300) then c300 else if not (nu\
ll c200) then c200 else if not (null c100) then c100 else c000

summarizeS s =
  let a = replace "\"" "'" $ concatMap (\x -> fst x ++ " ") $ summarize s in
  if not (null a) then a else safeFirst $ segment s where
    safeFirst x
      | length x > 1 = head x ++ x !! 1
      | not (null x)   = head x
      | otherwise    = ""

main = do
  let s = "Plunging European stocks, wobbly bonds and grave concerns about the h\
ealth of Portuguese lender Banco Espirito Santo SA made last week feel like a re\
run of the euro crisis, but most investors say it was no more than a blip for a \
resurgent region. Banco Espirito Santo has been in investorsâ€™ sights since Dec\
ember, when The Wall Street Journal first reported on accounting irregularities \
at the complex firm. Nerves frayed on Thursday when Banco Espirito Santo's paren\
t company said it wouldn't be able to meet some short-term debt obligations."
  print $ summarize s
  print $ summarizeS s
```

Lazy evaluation allows us in function **summarize** to define summaries of various numbers of sentences, but not all of these possible summaries are calculated.

```
$ stack ghci
*Main ... > :l Summarize.hs
*Summarize> main
[("Nerves frayed on Thursday when Banco Espirito Santo's parent company said it \
wouldn't be able to meet some short-term debt obligations.",193.54500000000002)]
"Nerves frayed on Thursday when Banco Espirito Santo's parent company said it wo\
uldn't be able to meet some short-term debt obligations. "
```

Part of Speech Tagging

We close out this chapter with the Haskell version of my part of speech (POS) tagger that I originally wrote in Common Lisp, then converted to Ruby and Java. The file *LexiconData.hs* is similar to the

lexical data files seen earlier: I am defining a map where keys a words and map values are POS tokens like *NNP* (proper noun), *RB* (adverb), etc. The file *README.md* contains a complete list of POS tag definitions.

The example code and data for this section is in the directory *FastTag*.

This listing shows a tiny representative part of the POS definitions in *LexiconData.hs*:

```
lexicon = M.fromList [("AARP", "NNP"), ("Clinic", "NNP"), ("Closed", "VBN"),
                      ("Robert", "NNP"), ("West-German", "JJ"),
                      ("afterwards", "RB"), ("arises", "VBZ"),
                                            ("attacked", "VBN"), ...]
```

Before looking at the code example listing, let's see how the functions defined in *fasttag.hs* work in a GHCi repl:

```
*Main LexiconData> bigram ["the", "dog", "ran",
                           "around", "the", "tree"]
[["the","dog"],["dog","ran"],["ran","around"],
 ["around","the"],["the","tree"]]
*Main LexiconData> tagHelper "car"
["car","NN"]
*Main LexiconData> tagHelper "run"
["run","VB"]
*Main LexiconData> substitute ["the", "dog", "ran", "around",
                               "the", "tree"]
[[["the","DT"],["dog","NN"]],[["dog","NN"],["ran","VBD"]],
 [["ran","VBD"],["around","IN"]],[["around","IN"],["the","DT"]],
 [["the","DT"],["tree","NN"]]]
*Main LexiconData> fixTags $  substitute ["the", "dog", "ran",
                                          "around", "the", "tree"]
["NN","VBD","IN","DT","NN"]
```

Function **bigram** takes a list or words and returns a list of word pairs. We need the word pairs because parts of the tagging algorithm needs to see a word with its preceeding word. In an imperative language, I would loop over the words and for a word at index **i** I would have the word at index **i - 1**. In a functional language, we avoid using loops and in this case create a list of adjacent word pairs to avoid having to use an explicit loop. I like this style of functional programming but if you come from years of using imperative language like Java and C++ it takes some getting used to.

tagHelper converts a word into a list of the word and its likely tag. **substitute** applies **tagHelper** to a list of words, getting the most probable tag for each word. The function **fixTags** will occasionally

override the default word tags based on a few rules that are derived from Eric Brill's paper A Simple Rule-Based Part of Speech Tagger[22].

Here is the entire example:

```
module Main where

import qualified Data.Map as M
import Data.Strings (strEndsWith, strStartsWith)
import Data.List (isInfixOf)

import LexiconData (lexicon)

bigram :: [a] -> [[a]]
bigram [] = []
bigram [_] = []
bigram xs = take 2 xs : bigram (tail xs)

containsString word substring = isInfixOf substring word

fixTags twogramList =
  map
  -- in the following inner function, [last,current] might be bound,
  -- for example, to [["dog","NN"],["ran","VBD"]]
  (\[last, current] ->
    -- rule 1: DT, {VBD | VBP} --> DT, NN
    if last !! 1 == "DT" && (current !! 1 == "VBD" ||
                             current !! 1 == "VB" ||
                             current !! 1 == "VBP")
    then "NN"
    else
      -- rule 2: convert a noun to a number (CD) if "." appears in the word
      if (current !! 1) !! 0 == 'N' && containsString (current !! 0) "."
      then "CD"
      else
        -- rule 3: convert a noun to a past participle if
        --         words.get(i) ends with "ed"
        if (current !! 1) !! 0 == 'N' && strEndsWith (current !! 0) "ed"
        then "VBN"
        else
          -- rule 4: convert any type to adverb if it ends in "ly"
          if strEndsWith (current !! 0) "ly"
```

[22] http://aclweb.org/anthology/A92-1021

```
38            then "RB"
39            else
40              -- rule 5: convert a common noun (NN or NNS) to an
41              --         adjective if it ends with "al"
42              if strStartsWith (current !! 1) "NN" &&
43                 strEndsWith (current !! 1) "al"
44              then "JJ"
45              else
46                -- rule 6: convert a noun to a verb if the preceeding
47                --         word is "would"
48                if strStartsWith (current !! 1) "NN" &&
49                   (last !! 0) == "would" -- should be case insensitive
50                then "VB"
51                else
52                  -- rule 7: if a word has been categorized as a
53                  --         common noun and it ends with "s",
54                  --         then set its type to plural common noun (NNS)
55                  if strStartsWith (current !! 1) "NN" &&
56                     strEndsWith (current !! 0) "s"
57                  then "NNS"
58                  else
59                    -- rule 8: convert a common noun to a present
60                    --         participle verb (i.e., a gerand)
61                    if strStartsWith (current !! 1) "NN" &&
62                       strEndsWith (current !! 0) "ing"
63                    then "VBG"
64                    else (current !! 1))
65   twogramList
66
67  substitute tks = bigram $ map tagHelper tks
68
69  tagHelper token =
70    let tags = M.findWithDefault [] token lexicon in
71    if tags == [] then [token, "NN"] else [token, tags]
72
73  tag tokens = fixTags $ substitute ([""] ++ tokens)
74
75
76  main = do
77    let tokens = ["the", "dog", "ran", "around", "the", "tree", "while",
78                  "the", "cat", "snaked", "around", "the", "trunk",
79                  "while", "banking", "to", "the", "left"]
```

```
80    print $ tag tokens
81    print $ zip tokens $ tag tokens
```

```
*Main LexiconData> main
["DT","NN","VBD","IN","DT","NN","IN","DT","NN","VBD","IN","DT",
 "NN","IN","VBG","TO","DT","VBN"]
[("the","DT"),("dog","NN"),("ran","VBD"),("around","IN"),
 ("the","DT"),("tree","NN"),("while","IN"),("the","DT"),
 ("cat","NN"),("snaked","VBD"),("around","IN"),("the","DT"),
 ("trunk","NN"),("while","IN"),("banking","VBG"),("to","TO"),
 ("the","DT"),("left","VBN")]
```

The README.md file contains definitions of the POS definitions. Here are the ones used in this example:

```
DT Determiner               the,some
NN noun                     dog,cat,road
VBD verb, past tense        ate,ran
IN Preposition              of,in,by
```

Natural Language Processing Wrap Up

NLP is a large topic. I have attempted to show you just the few tricks that I use often and are simple to implement. I hope that you reuse the code in this chapter in your own projects when you need to detect entities, classify text, summarize text, and assign part of speech tags to words in text.

Linked Data and the Semantic Web

I am going to show you how to query semantic web data sources on the web and provide examples for how you might use this data in applications. I have written two previous books on the semantic web, one covering Common Lisp and the other covering JVM languages Java, Scala, Clojure, and Ruby. You can get fee PDF versions on the book page of www.makwatson.com[23]. If you enjoy the lite introduction in this chapter then please do download a free copy of my semantic web book for more material on RDF, RDFS, and SPARQL.

I like to think of the semantic web and linked data resources as:

- A source of structured data on the web. These resources are called SPARQL endpoints.
- Data is represented by data triples: subject, predicate, and object. The subject of one triple can be the object of another triple. Predicates are relationships; a few examples: "owns", "is part of", "author of", etc.
- Data that is accessed via the SPARQL query language.
- A source of data that may or may not be available. SPARQL endpoints are typically available for free use and they are sometimes unavailable. Although not covered here, I sometimes work around this problem by adding a caching layer to SPARQL queries (access key being a SPARQL query string, the value being the query results). This caching speeds up development and running unit tests, and sometimes saves a customer demo when a required SPARQL endpoint goes offline at an inconvenient time.

DBPedia is the semantic web version of Wikipedia[24]. The many millions of data triples that make up DBPedia are mostly derived from the structured "info boxes" on Wikipedia pages.

As you are learning SPARQL use the DBPedia SPARQL endpoint[25] to practice. As a practitioner who uses linked data, for any new project I start by identifying SPARQL endpoints for possibly useful data. I then interactively experiment with SPARQL queries to extract the data I need. Only when I am satisfied with the choice of SPARQL endpoints and SPARQL queries do I write any code to automatically fetch linked data for my application.

Pro tip: I mentioned SPARQL query caching. I sometimes cache query results in a local database, saving the returned RDF data indexed by the SPARQL query. You can also store the cache timestamp and refresh the cache every few weeks as needed. In addition to making development and unit testing faster, your applications will be more resilient.

[23] http://www.markwatson.com/books/
[24] http://wiki.dbpedia.org/
[25] http://dbpedia.org/sparql

In the last chapter "Natural Language Processing Tools" we resolved entities in natural language text to DBPedia (semantic web SPAQL endpoint for Wikipedia) URIs. Here we will use some of these URIs to demonstrate fetching real world knowledge that you might want to use in applications.

The SPARQL Query Language

Example RDF N3 triples (subject, predicate, object) might look like:

```
<http://www.markwatson.com>
  <http://dbpedia.org/ontology/owner>
  "Mark Watson" .
```

Element of triples can be URIs or string constants. Triples are often written all on one line; I split it to three lines to fit the page width. Here the subject is the URI for my web site, the predicate is a URI defining an ownership relationship, and the object is a string literal.

If you want to see details for any property or other URI you see, then "follow your nose" and open the URI in a web browser. For example remove the brackets from the owner property URI http://dbpedia.org/ontology/owner[26] and open it in a web browser. For working with RDF data programatically, it is convenient using full URI. For humans reading RDF, the N3 notation is better because it supports defining URI standard prefixes for use as abbreviations; for example:

```
prefix ontology: <http://dbpedia.org/ontology/>

<http://www.markwatson.com>
  ontology:owner
  "Mark Watson" .
```

If you wanted to find all things that I own (assuming this data was in a public RDF repository, which it isn't) then we might think to match the pattern:

```
prefix ontology: <http://dbpedia.org/ontology/>

?subject ontology:owner "Mark Watson"
```

And return all URIs matching the variable **?subject** as the query result. This is the basic idea of making SPARQL queries.

The following SPARQL query will be implemented later in Haskell using the HSparql library:

[26] http://dbpedia.org/ontology/owner

```
prefix resource: <http://dbpedia.org/resource/>
prefix dbpprop: <http://dbpedia.org/property/>
prefix foaf: <http://xmlns.com/foaf/0.1/>

SELECT *
WHERE {
    ?s dbpprop:genre resource:Web_browser .
    ?s foaf:name ?name .
} LIMIT 5
```

In this last SPARQL query example, the triple patterns we are trying to match are inside a *WHERE* clause. Notice that in the two triple patterns, the subject field of each is the variable **?s**. The first pattern matches all DBPedia triples with a predicate http://dbpedia.org/property/genre and an object equal to http://dbpedia.org/resource/Web_browser. We then find all triples with the same subject but with a predicate equal to http://xmlns.com/foaf/0.1/name.

Each result from this query will contain two values for variables **?s** and **?name**: a DBPedia URI for some thing and the name for that thing. Later we will run this query using Haskell code and you can see what the output might look like.

Sometimes when I am using a specific SPARQL query in an application, I don't bother defining prefixes and just use URIs in the query. As an example, suppose I want to return the Wikipedia (or DBPedia) abstract for IBM. I might use a query such as:

```
select * where {
  <http://dbpedia.org/resource/IBM>
  <http://dbpedia.org/ontology/abstract>
  ?o .
  FILTER langMatches(lang(?o), "EN")
} LIMIT 100
```

If you try this query using the web interface for DBPedia SPARQL queries[27] you get just one result because of the FILTER option that only returns English language results. You could also use FR for French results, GE for German results, etc.

A Haskell HTTP Based SPARQL Client

One approach to query the DBPedia SPARQL endpoint is to build a HTTP GET request, send it to the SPARQL endpoint server, and parse the returned XML response. We will start with this simple approach. You will recognize the SPARQL query from the last section:

[27] http://dbpedia.org/sparql/

```
1  {-# LANGUAGE OverloadedStrings #-}
2
3  module HttpSparqlClient where
4
5  import Network.HTTP.Conduit (simpleHttp)
6  import Network.HTTP.Base (urlEncode)
7  import Text.XML.HXT.Core
8  import Text.HandsomeSoup
9  import qualified Data.ByteString.Lazy.Char8 as B
10
11 buildQuery :: String -> [Char]
12 buildQuery sparqlString =
13   "http://dbpedia.org/sparql/?query=" ++ urlEncode sparqlString
14
15 main :: IO ()
16 main = do
17   let query = buildQuery "select * where {<http://dbpedia.org/resource/IBM> <htt\
18 p://dbpedia.org/ontology/abstract> ?o . FILTER langMatches(lang(?o), \"EN\")} LI\
19 MIT 100"
20   res <- simpleHttp query
21   let doc = readString []  (B.unpack res)
22   putStrLn "\nAbstracts:\n"
23   abstracts <- runX $ doc >>> css "binding" >>>
24                              (getAttrValue "name" &&& (deep getText))
25   print abstracts
```

The function **buildQuery** defined in lined 11-13 takes any SPARQL query, URL encodes it so it can be passed as part of a URI, and builds a query string for the DBPedia SPARQL endpoint. The returned data is in XML format. In lines 23-24 I am using the **XHT** parsing library to extract the names (values bound to the variable **?o** in the query in line 17). I cover the use of **HXT** and the **HandsomeSoup** parsing libraries in the next chapter.

In the **main** function, we use the utility function **simpleHttp** in line 20 to fetch the results as a ByteString and in line 21 we unback this to a regular Haskell String.

```
Prelude> :l HttpSparqlClient.hs
[1 of 1] Compiling HttpSparqlClient ( HttpSparqlClient.hs, interpreted )
Ok, modules loaded: HttpSparqlClient.
*HttpSparqlClient> main

Abstracts:

[("o","International Business Machines Corporation (commonly referred to as IBM)\
 is an American multinational technology and consulting corporation, with corpor\
ate headquarters in Armonk, New York.
  ...)]
```

Querying Remote SPARQL Endpoints

We will write some code in this section to make the example query to get the names of web browsers from DBPedia. In the last section we made a SPARQL query using fairly low level Haskell libraries. We will be using the high level library *HSparql* to build SPARQL queries and call the DBPedia SPARQL endpoint.

The example in this section can be found in *SparqlClient/TestSparqlClient.hs*. In the **main** function notice how I have commented out printouts of the raw query results. Because Haskell is type safe, extracting the values wrapped in query results requires knowing RDF element return types. I will explain this matching after the program listing:

```
-- simple experiments with the excellent HSparql library

module Main where

import Database.HSparql.Connection (BindingValue(Bound))

import Data.RDF hiding (triple)
import Database.HSparql.QueryGenerator
import Database.HSparql.Connection (selectQuery)

webBrowserSelect :: Query SelectQuery
webBrowserSelect = do
    resource <- prefix "dbprop" (iriRef "http://dbpedia.org/resource/")
    dbpprop  <- prefix "dbpedia" (iriRef "http://dbpedia.org/property/")
    foaf     <- prefix "foaf" (iriRef "http://xmlns.com/foaf/0.1/")
    x    <- var
    name <- var
    triple x (dbpprop .:. "genre") (resource .:. "Web_browser")
```

```
    triple x (foaf .:. "name") name

    return SelectQuery { queryVars = [name] }

companyAbstractSelect :: Query SelectQuery
companyAbstractSelect = do
    resource <- prefix "dbprop" (iriRef "http://dbpedia.org/resource/")
    ontology <- prefix "ontology" (iriRef "http://dbpedia.org/ontology/")
    o <- var
    triple (resource .:. "Edinburgh_University_Press") (ontology .:. "abstract")\
 o
    return SelectQuery { queryVars = [o] }

companyTypeSelect :: Query SelectQuery
companyTypeSelect = do
    resource <- prefix "dbprop" (iriRef "http://dbpedia.org/resource/")
    ontology <- prefix "ontology" (iriRef "http://dbpedia.org/ontology/")
    o <- var
    triple (resource .:. "Edinburgh_University_Press") (ontology .:. "type") o
    return SelectQuery { queryVars = [o] }

main :: IO ()
main = do
  sq1 <- selectQuery "http://dbpedia.org/sparql" companyAbstractSelect
  --putStrLn "\nRaw results of company abstract SPARQL query:\n"
  --print sq1
  putStrLn "\nWeb browser names extracted from the company abstract query result\
s:\n"
  case sq1 of
    Just a -> print $ map (\[Bound (LNode (PlainLL s _))] -> s) a
    Nothing -> putStrLn "nothing"
  sq2 <- selectQuery "http://dbpedia.org/sparql" companyTypeSelect
  --putStrLn "\nRaw results of company type SPARQL query:\n"
  --print sq2
  putStrLn "\nWeb browser names extracted from the company type query results:\n"
  case sq2 of
    Just a -> print $ map (\[Bound (UNode  s)] -> s) a
    Nothing -> putStrLn "nothing"
  sq3 <- selectQuery "http://dbpedia.org/sparql" webBrowserSelect
  --putStrLn "\nRaw results of SPARQL query:\n"
  --print sq3
  putStrLn "\nWeb browser names extracted from the query results:\n"
```

```
  case sq3 of
    Just a -> print $ map (\[Bound (LNode (PlainLL s _))] -> s) a
    Nothing -> putStrLn "nothing"
```

Notes on matching result types of query results:

You will notice how I have commented out print statements in the last example. When trying new queries you need to print out the results in order to know how to extract the wrapped query results. Let's look at a few examples:

If we print the value for **sq1**:

```
Raw results of company abstract SPARQL query:

Just [[Bound (LNode (PlainLL "Edinburgh University Press ...
```

we see that inside a **Just** we have a list of lists. Each inner list is a **Bound** wrapping types defined in HSparql. We would unwrap **sq1** using:

```
  case sq1 of
    Just a -> print $ map (\[Bound (LNode (PlainLL s _))] -> s) a
    Nothing -> putStrLn "nothing"
```

In a similar way I printed out the values of **sq2** and **sq3** to see the form os **case** statement I would need to unwrap them.

The output from this example with three queries to the DBPedia SPARQL endpoint is:

```
Web browser names extracted from the company abstract query results in sq1:

["Edinburgh University Press \195\168 una casa editrice scientifica di libri acc\
ademici e riviste, con sede a Edimburgo, in Scozia.","Edinburgh University Press\
 \195\169 uma editora universit\195\161ria com base em Edinburgh, Esc\195\179cia\
.","Edinburgh University Press is a scholarly publisher of academic books and jo\
urnals, based in Edinburgh, Scotland."]

The type of company is extracted from the company type query results in sq2:

["http://dbpedia.org/resource/Publishing"]

Web browser names extracted from the query results in sq3:

["Grail","ViolaWWW","Kirix Strata","SharkWire Online","MacWeb","Camino","eww","T\
```

```
enFourFox","WiseStamp","X-Smiles","Netscape Navigator 2","SimpleTest","AWeb","IB\
rowse","iCab","ANT Fresco","Netscape Navigator 9.0","HtmlUnit","ZAC Browser","EL\
inks","ANT Galio","Nintendo DSi Browser","Nintendo DS Browser","Netscape Navigat\
or","NetPositive","OmniWeb","Abaco","Flock","Steel","Kazehakase","GNU IceCat","F\
reeWRL","UltraBrowser","AMosaic","NetCaptor","NetSurf","Netscape Browser","SlipK\
not","ColorZilla","Internet Channel","Obigo Browser","Swiftfox","BumperCar","Swi\
ftweasel","Swiftdove","IEs4Linux","MacWWW","IBM Lotus Symphony","SlimBrowser","c\
URL","FoxyTunes","Iceweasel","MenuBox","Timberwolf web browser","Classilla","Roc\
kmelt","Galeon","Links","Netscape Navigator","NCSA Mosaic","MidasWWW","w3m","Poi\
nterWare","Pogo Browser","Oregano","Avant Browser","Wget","NeoPlanet","Voyager",\
"Amaya","Midori","Sleipnir","Tor","AOL Explorer"]
```

Linked Data and Semantic Web Wrap Up

If you enjoyed the material on linked data and DBPedia then please do get a free copy of one of my semantic web books on my website book page[28] as well as other SPARQL and linked data tutorials on the web.

Structured and sematically labelled data, when it is available, is much easier to process and use effectively than raw text and HTML collected from web sites.

[28] http://www.markwatson.com/books/

Web Scraping

In my past work I usually used the Ruby scripting language for web scraping but as I use the Haskell language more often for projects both large and small I am now using Haskell for web scraping, data collection, and data cleaning tasks. If you worked through the tutorial chapter on impure Haskell programming then you already know most of what you need to understand this chapter. Here we will walk through a few short examples for common web scraping tasks.

Before we start a tutorial about web scraping I want to point out that much of the information on the web is copyright and the first thing that you should do is to read the terms of service for web sites to insure that your use of web scraped data conforms with the wishes of the persons or organizations who own the content and pay to run scraped web sites.

As we saw in the last chapter on linked data there is a huge amount of structured data available on the web via web services, semantic web/linked data markup, and APIs. That said, you will frequently find text (usually HTML) that is useful on web sites. However, this text is often at least partially unstructured and in a messy and frequently changing format because web pages are meant for human consumption and making them easy to parse and use by software agents is not a priority of web site owners.

Note: It takes a while to fetch all of the libraries in the directory *WebScraping* so please do a **stack build** now to get these examples ready to experiment with while you read this chapter.

Using the Wreq Library

The *Wreq* library[29] is an easy way to fetch data from the web. The example in this section fetches DBPedia (i.e., the semantic web version of Wikipedia) data in JSON and RDF N3 formats, and also fetches the index page from my web site. I will introduce you to the *Lens* library for extracting data from data structures, and we will also use *Lens* in a later chapter when writing a program to play Backjack.

We will be using function **get** in the **Network.Wreq** module that has a type signature:

```
get
  :: String -> IO (Response Data.ByteString.Lazy.Internal.ByteString)
```

We will be using the **OverloadedStrings** language extension to facilitate using both **[Char]** strings and **ByteString** data types. Note: In the GHCi repl you can use **:set -XOverloadedStrings**.

We use function **get** to return JSON data; here is a bit of the JSON data returned from calling **get** using the URI for my web site:

[29] http://www.serpentine.com/wreq/tutorial.html

```
Response {responseStatus = Status {statusCode = 200, statusMessage = "OK"},
          responseVersion = HTTP/1.1,
          responseHeaders =
                    [("Date","Sat, 15 Oct 2016 16:00:59 GMT"),
                     ("Content-Type","text/html"),
                ("Transfer-Encoding","chunked"),
                ("Connection","keep-alive")],
          responseBody = "<!DOCTYPE html>\r\n<html>\r\n<head><title>Mark Watson:\
 consultant specializing in artificial intelligence, natural language processing\
, and machine\r\n    learning</title>\r\n    <meta name=\"viewport\" content=\"w\
idth=device-width, initial-scale=1.0\">\r\n    <meta name=\"msvalidate.01\" cont\
ent=\"D980F894E94AA6335FB595676DFDD5E6\"/>\r\n    <link href=\"/css/bootstrap.mi\
n.css\" rel=\"stylesheet\" type=\"text/css\">\r\n    <link href=\"/css/bootstrap\
-theme.min.css\" rel=\"stylesheet\" type=\"text/css\">\r\n    <link href=\"/css/\
mark.css\" rel=\"stylesheet\" type=\"text/css\">\r\n    <link rel=\"manifest\" h\
ref=\"/manifest.json\">\r\n    <style type=\"text/css\">
          body {\r\n        padding-top: 60px;\r\n    }</style>\r\n\r\n    <link\
 rel=\"canonical\" href=https://www.markwatson.com/ />\r\n</head>\r\n<body  href\
=\"http://blog.markwatson.com\">Blog</a></li>\r\n
          <li class=\"\"><a href=\"/books/\">My Books</a>
```

As an example, the *Lens* expression for extracting the response status code is (**r** is the **IO Response** data returned from calling **get**):

```
(r ^. responseStatus . statusCode)
```

responseStatus digs into the top level response structure and **statusCode** digs further in to fetch the code 200. To get the actual contents of the web page we can use the **responseBody** function:

```
(r ^. responseBody)
```

Here is the code for the entire example:

```
{-# LANGUAGE OverloadedStrings #-}

-- reference: http://www.serpentine.com/wreq/tutorial.html

module HttpClientExample where

import Network.Wreq
import Control.Lens  -- for ^. ^?
import Data.Maybe (fromJust)

fetchURI uri = do
  putStrLn $ "\n\n***  Fetching " ++ uri
  r <- get uri
  putStrLn $ "status code: " ++ (show (r ^. responseStatus . statusCode))
  putStrLn $ "content type: " ++ (show (r ^? responseHeader "Content-Type"))
  putStrLn $ "respose body: " ++ show (fromJust (r ^? responseBody))

main :: IO ()
main = do
  -- JSON from DBPedia
  fetchURI "http://dbpedia.org/data/Sedona_Arizona.json"
  -- N3 RDF from DBPedia
  fetchURI "http://dbpedia.org/data/Sedona_Arizona.n3"
  -- my web site
  fetchURI "http://markwatson.com"
```

This example produces a lot of printout, so I a just showing a small bit here (the text from the body is not shown):

```
*Main> :l HttpClientExample
[1 of 1] Compiling HttpClientExample ( HttpClientExample.hs, interpreted )
Ok, modules loaded: HttpClientExample.
*HttpClientExample> main

***  Fetching http://dbpedia.org/data/Sedona_Arizona.json
status code: 200
content type: Just "application/json"
respose body: "{\n  \"http://en.wikipedia.org/wiki/Sedona_Arizona\" : { \"http:/\
/xmlns.com/foaf/0.1/primaryTopic\" : [ { \"type\" : \"uri\", \"value\" : \"http:\
//dbpedia.org/resource/Sedona_Arizona\" } ] } ,\n  \"http://dbpedia.org/resource\
/Sedona_Arizona\" : { \"http://www.w3.org/2002/07/owl#sameAs\" : [ { \"type\" : \
\"uri\", \"value\" : \"http://dbpedia.org/resource/Sedona_Arizona\" } ] ,\n    \\
```

```
"http://www.w3.org/2000/01/rdf-schema#label\" : [ { \"type\" : \"literal\", \"va\
lue\" : \"Sedona Arizona\" , \"lang\" : \"en\" } ] ,\n    \"http://xmlns.com/foa\
f/0.1/isPrimaryTopicOf\" : [ { \"type\" : \"uri\", \"value\" : \"http://en.wikip\
edia.org/wiki/Sedona_Arizona\" } ] ,\n    \"http://www.w3.org/ns/prov#wasDerived\
From\" : [ { \"type\" : \"uri\", \"value\" : \"http://en.wikipedia.org/wiki/Sedo\
na_Arizona?oldid=345939723\" } ] ,\n    \"http://dbpedia.org/ontology/wikiPageID\
\" : [ { \"type\" : \"literal\", \"value\" : 11034313 , \"datatype\" : \"http://\
www.w3.org/2001/XMLSchema#integer\" } ] ,\n    \"http://dbpedia.org/ontology/wik\
iPageRevisionID\" : [ { \"type\" : \"literal\", \"value\" : 345939723 , \"dataty\
pe\" : \"http://www.w3.org/2001/XMLSchema#integer\" } ] ,\n    \"http://dbpedia.\
org/ontology/wikiPageRedirects\" : [ { \"type\" : \"uri\", \"value\" : \"http://\
dbpedia.org/resource/Sedona,_Arizona\" } ] }\n}\n"

***  Fetching http://dbpedia.org/data/Sedona_Arizona.n3
status code: 200
content type: Just "text/n3; charset=UTF-8"
respose body: "@prefix foaf:\t<http://xmlns.com/foaf/0.1/> .\n@prefix wikipedia-\
en:\t<http://en.wikipedia.org/wiki/> .\n@prefix dbr:\t<http://dbpedia.org/resour\
ce/> .\nwikipedia-en:Sedona_Arizona\tfoaf:primaryTopic\tdbr:Sedona_Arizona .\n@p\
refix owl:\t<http://www.w3.org/2002/07/owl#> .\ndbr:Sedona_Arizona\towl:sameAs\t\
dbr:Sedona_Arizona .\n@prefix rdfs:\t<http://www.w3.org/2000/01/rdf-schema#> .\n\
dbr:Sedona_Arizona\trdfs:label\t\"Sedona Arizona\"@en ;\n\tfoaf:isPrimaryTopicOf\
\twikipedia-en:Sedona_Arizona .\n@prefix prov:\t<http://www.w3.org/ns/prov#> .\n\
dbr:Sedona_Arizona\tprov:wasDerivedFrom\t<http://en.wikipedia.org/wiki/Sedona_Ar\
izona?oldid=345939723> .\n@prefix dbo:\t<http://dbpedia.org/ontology/> .\ndbr:Se\
dona_Arizona\tdbo:wikiPageID\t11034313 ;\n\tdbo:wikiPageRevisionID\t345939723 ;\\
n\tdbo:wikiPageRedirects\t<http://dbpedia.org/resource/Sedona,_Arizona> ."

***  Fetching http://markwatson.com
status code: 200
content type: Just "text/html"
respose body: "<!DOCTYPE html>\r\n<html>\r\n<head><title>Mark Watson: consultant\
 specializing in  ...
```

You might want to experiment in the GHCi repl with the **get** function and *Lens*. If so, this will get you started:

```
*Main> :set -XOverloadedStrings
*Main> r <- get "http://dbpedia.org/data/Sedona_Arizona.json"
*Main> :t r
r :: Response ByteString
*Main> (r ^. responseStatus . statusCode)
200
*Main> (r ^? responseHeader "Content-Type")
Just "application/json"
*Main> fromJust (r ^? responseHeader "Content-Type")
"application/json"
*Main> (fromJust (r ^? responseBody))
"{\n  \"http://en.wikipedia.org/wiki/Sedona_Arizona\" : { ... not shown ... \"
```

In the following section we will use the *HandsomeSoup* library for parsing HTML.

Using the HandsomeSoup Library for Parsing HTML

We will now use the Handsome Soup[30] library to parse HTML. Handsome Soup allows us to use CSS style selectors to extract specific elements from the HTML from a web page. The HXT lower level library provides modeling HTML (and XML) as a tree structure and an *Arrow*[31] style interface for traversing the tree structures and extract data. Arrows are a generalization of monads to manage calculations given a context. I will touch upon just enough material on Arrows for you to understand the examples in this chapter. Handsome Soup also provides a high level utility function **fromUrl** to fetch web pages; the type of **fromUrl** is:

```
fromUrl
  :: String -> IOSArrow b (Data.Tree.NTree.TypeDefs.NTree XNode)
```

We will not work directly with the tree structure of the returned data, we will simply use the accessor functions to extract the data we need. Before looking at the example code listing, let's look at this extraction process (**doc** is the tree structured data returned from calling **fromUrl**):

```
  links <- runX $ doc >>> css "a" ! "href"
```

The **runX** function runs arrow computations for us. **doc** is a tree data structure, **css** allows us to pattern match on specific HTML elements.

Here we are using CSS style selection for all "a" anchor HTML elements and digging into the element to return the element attribute "href" value for each "a" anchor element. In a similar way, we can select all "img" image elements and dig down into the matched elements to fetch the "src" attributes:

[30] https://github.com/egonSchiele/HandsomeSoup
[31] https://wiki.haskell.org/Arrow

```
  imageSrc <- runX $ doc >>> css "img" ! "src"
```

We can get the full body text:

```
  allBodyText <- runX $ doc >>> css "body" //> getText
```

The operator //> applied to the function **getText** will get all text in all nested elements inside the *body* element. If we had used the operator /> then we would only have fetched the text at the top level of the body element.

Here is the full example source listing:

```
{-# LANGUAGE OverloadedStrings #-}

-- references: https://github.com/egonSchiele/HandsomeSoup
--             http://adit.io/posts/2012-04-14-working_with_HTML_in_haskell.html

module Main where

import Text.XML.HXT.Core
import Text.HandsomeSoup


main :: IO ()
main = do
  let doc = fromUrl "http://markwatson.com/"
  putStrLn "\n\n ** LINKS:\n"
  links <- runX $ doc >>> css "a" ! "href"
  mapM_ putStrLn links
  h2 <- runX $ doc >>> css "h2" ! "href"
  putStrLn "\n\n ** ALL H2 ELEMENTS::\n"
  mapM_ putStrLn h2
  imageSrc <- runX $ doc >>> css "img" ! "src"
  putStrLn "\n\n ** ALL IMG ELEMENTS:\n"
  mapM_ putStrLn imageSrc
  allBodyText <- runX $ doc >>> css "body" //> getText
  putStrLn "\n\n ** TEXT FROM BODY ELEMENT:\n"
  mapM_ putStrLn allBodyText
  pText <- runX $ doc >>> css "p" //> getText -- //> gets all contained text
                                             -- />  gets only directly
                                             --     contained text
  putStrLn "\n\n ** ALL P ELEMENTS:\n"
  mapM_ putStrLn pText
```

This example prints out several hundred lines; here is the first bit of output:

```
*Main> :l HandsomeSoupTest.hs
[1 of 1] Compiling HandsomeSoupTest ( HandsomeSoupTest.hs, interpreted )
Ok, modules loaded: HandsomeSoupTest.
*HandsomeSoupTest> main
 ** LINKS:
/
/consulting/
http://blog.markwatson.com
/books/
/opensource/
/fun/
https://github.com/mark-watson
https://plus.google.com/117612439870300277560
https://twitter.com/mark_l_watson
https://www.wikidata.org/wiki/Q18670263
http://markwatson.com/index.rdf
http://markwatson.com/index.ttl

 ** ALL IMG ELEMENTS:
/pictures/Markws.jpg

 ** TEXT FROM BODY ELEMENT:
   ...
```

I find HandsomeSoup to be very convenient for picking apart HTML data fetched from web pages. Writing a good spider for any given web site is a process of understanding how the HTML for the web site is structured and what information you need to collect. I strongly suggest that you work with the web page to be spider open in a web browser with "show source code" in another browser tab. Then open an interactive GHCi repl and experiment using the HandsomeSoup APIs to get the data you need.

Web Scraping Wrap Up

There are many Haskell library options for web scraping and cleaning data. In this chapter I showed you just what I use in my projects.

The material in this chapter and the chapters on text processing and linked data should be sufficient to get you started using online data sources in your applications.

Using Relational Databases

We will see how to use popular libraries for accessing the *sqlite* and *Postgres* (sometimes also called *PostgeSQL*) databases in this chapter. I assume that you are already familiar with *SQL*.

Database Access for Sqlite

We will use the sqlite-simple[32] library in this section to access Sqlite databases and use the similar library postgresql-simple[33] in the next section for use with Postgres.

There are other good libraries for database connectivity like Persistent[34] but I like sqlite-simple and it has a gentle learning curve so that is what we will use here. You will learn the basics of database connectivity in this and the next section. Setting up and using *sqlite* is easy because the *sqlite-simple* library includes the compiled code for *sqlite* so configuration requires only the file path to the database file.

```
1  {-# LANGUAGE OverloadedStrings #-}
2
3  module Main where
4
5  import Database.SQLite.Simple
6
7  {-
8     Create sqlite database:
9       sqlite3 test.db "create table test (id integer primary key, str text);"
10
11    This example is derived from the example at github.com/nurpax/sqlite-simple
12 -}
13
14 main :: IO ()
15 main = do
16   conn <- open "test.db"
17   -- start by getting table names in database:
18   do
19     r <- query_ conn
```

[32] https://hackage.haskell.org/package/sqlite-simple
[33] https://hackage.haskell.org/package/postgresql-simple
[34] https://www.stackage.org/package/persistent

```
20        "SELECT name FROM sqlite_master WHERE type='table'" :: IO [Only String]
21      print "Table names in database test.db:"
22      mapM_ (print . fromOnly) r
23
24    -- get the metadata for table test in test.db:
25    do
26      r <- query_ conn
27            "SELECT sql FROM sqlite_master WHERE type='table' and name='test'" ::
28                  IO [Only String]
29      print "SQL to create table 'test' in database test.db:"
30      mapM_ (print . fromOnly) r
31
32    -- add a row to table 'test' and then print out the rows in table 'test':
33    do
34      execute conn "INSERT INTO test (str) VALUES (?)"
35        (Only ("test string 2" :: String))
36      r2 <- query_ conn "SELECT * from test" :: IO [(Int, String)]
37      print "number of rows in table 'test':"
38      print (length r2)
39      print "rows in table 'test':"
40      mapM_ print  r2
41
42    close conn
```

The type **Only** used in line 20 acts as a container for a single value and is defined in the *simple-sqlite* library. It can also be used to pass values for queries like:

```
r <- query_ conn "SELECT name FROM customers where id = ?" (Only 4::Int)
```

To run this example start by creating a sqlite database that is stored in the file *test.db*:

```
sqlite3 test.db "create table test (id integer primary key, str text);"
```

Then build and run the example:

```
stack build --exec TestSqLite1
```

Database Access for Postgres

Setting up and using a database in the last section was easy because the *sqlite-simple* library includes the compiled code for *sqlite* so configuration only requires the file path the the database file. The

Haskel examples for Postgres will be similar to those for Sqlite. There is some complication in setting up Postgres if you do not already have it installed and configured.

In any case, you will need to have Postgres installed and set up with a user account for yourself. When I am installing and configuring Postgres on my Linux laptop, I create a database role **markw**. You will certainly create a different role/account name so subsitute your role name for **markw** in the following code examples.

If you are using Ubuntu you can install Postgres and create a role using:

```
sudo apt-get update
sudo apt-get install postgresql postgresql-contrib postgresql-server-dev-9.5
sudo -u postgres createuser --interactive
Enter name of role to add: markw
Shall the new role be a superuser? (y/n) y
```

We will need to install postgresql-server-dev-9.5 in order to use the Haskell Postgres bindings. Note that your version of Ubuntu Linux may have a different version of the server dev package which you can find using:

```
aptitude search postgresql-dev
```

If you are using Mac OS X you can then install Postgres as an application which is convenient for development. A role is automatically created with the same name as your OS X "short name." You can use the "Open psql" button on the interface to open a command line shell that functions like the *psql* command on Ubuntu (or other Linux distributions).

We will need to install postgresql-server-dev-9.5 in order to use the Haskell Postgres bindings. Note that your version of Ubuntu Linux may have a different version of the server dev package which you can find using:

```
aptitude search postgresql-dev
```

You will then want to create a database named **haskell** and set the password for role/account **markw** to **test1** for running the example in this section:

```
createdb haskell
sudo -u postgres psql
postgres=# alter user markw encrypted password 'test1';
postgres=# \q

psql -U markw haskell
psql (9.5.4)
Type "help" for help.

haskell=# create table customers (id int, name text, email text);
CREATE TABLE
haskell=#  insert into customers values (1, 'Acme Cement', 'info@acmecement.com'\
);
INSERT 0 1
haskell=# \q
```

If you are not familiar with using Postgres then take a minute to experiment with using the *psql* command line utility to connect to the database you just created and peform practice queries:

```
markw=# \c haskell
You are now connected to database "haskell" as user "markw".
haskell=# \d
         List of relations
 Schema |   Name    | Type  | Owner
--------+-----------+-------+-------
 public | customers | table | markw
 public | links     | table | markw
 public | products  | table | markw
(3 rows)

haskell=# select * from customers;
 id |      name       |        email
----+-----------------+---------------------
  1 | Acme Cement     | info@acmecement.com
  2 | Biff Home Sales | info@biff.com
  3 | My Pens         | info@mypens.com
(3 rows)

haskell=# select * from products;
 id |     name      | cost
----+---------------+------
  1 | Cement bag    |  2.5
```

```
  2 | Cheap Pen     |  1.5
  3 | Expensive Pen | 14.5
(3 rows)

haskell=# select * from links;
 id | customer_id | productid
----+-------------+-----------
  1 |           1 |         1
  2 |           3 |         2
  3 |           3 |         3
(3 rows)

haskell=#
```

You can change default database settings using **ConnectInfo**:

```
ConnectInfo
  connectHost :: String
  connectPort :: Word16
  connectUser :: String
  connectPassword :: String
  connectDatabase :: String
```

In the following example on lines 9-10 I use **defaultConnectInfo** that lets me override just some settings, leaving the rest set at default values. The code to access a database using *simple-postgresql* is similar to that in the last section, with a few API changes.

```
1  {-# LANGUAGE OverloadedStrings #-}
2
3  module Main where
4
5  import Database.PostgreSQL.Simple
6
7  main :: IO ()
8  main = do
9    conn <- connect defaultConnectInfo { connectDatabase = "haskell",
10                                         connectUser = "markw" }
11   -- start by getting table names in database:
12   do
13     r <- query_ conn "SELECT name FROM customers" :: IO [(Only String)]
14     print "names and emails in table 'customers' in database haskell:"
15     mapM_ (print . fromOnly) r
```

```
16
17    -- add a row to table 'test' and then print out the rows in table 'test':
18    do
19      let rows :: [(Int, String, String)]
20          rows = [(4, "Mary Smith", "marys@acme.com")]
21      executeMany conn
22        "INSERT INTO customers (id, name, email) VALUES (?,?,?)" rows
23      r2 <- query_ conn "SELECT * from customers" :: IO [(Int, String, String)]
24      print "number of rows in table 'customers':"
25      print (length r2)
26      print "rows in table 'customers':"
27      mapM_ print  r2
28
29    close conn
```

The type **Only** used in line 20 acts as a container for a single value and is defined in the *simple-postgresql* library. It can also be used to pass values for queries like:

```
r <- query_ conn "SELECT name FROM customers where id = ?" (Only 4::Int)
```

The monad mapping function **mapM_** using in line 22 is like **mapM** but is used when we do not need the resulting collection from executing the map operation. **mapM_** is used for side effects, in this case extracting the value for a collection of **Only** values and printing them. I removed some output from building the example in the following listing:

```
$ Database-postgres git:(master) > stack build --exec TestPostgres1
TestDatabase-0.1.0.0: build
Preprocessing executable 'TestPostgres1' for TestDatabase-0.1.0.0...
[1 of 1] Compiling Main             ( TestPostgres1.hs,

"names and emails in table 'customers' in database haskell:"
"Acme Cement"
"Biff Home Sales"
"My Pens"
"number of rows in table 'customers':"
4
"rows in table 'customers':"
(1,"Acme Cement","info@acmecement.com")
(2,"Biff Home Sales","info@biff.com")
(3,"My Pens","info@mypens.com")
(4,"Mary Smith","marys@acme.com")
```

Postgres is my default database and I use it unless there is a compelling reason not to. While work for specific customers has mandated using alternative data stores (e.g., BigTable while working at Google and MongoDB at Compass Labs), Postgres supports relational tables, free text search, and structured data like JSON.

Haskell Program to Play the Blackjack Card Game

For much of my work using Haskell I deal mostly with pure code with smaller bits of impure code for network and file IO, etc. Realizing that my use case for using Haskell (mostly pure code) may not be typical, I wanted the last example "cookbook recipe" in this book to be an example dealing with changing state, a program to play the Blackjack card game.

The game state is maintained in the type **Table** that holds information on a randomized deck of cards, the number of players in addition to the game user and the card dealer, the cards in the current hand, and the number of betting chips that all players own. Table data is immutable so all of the major game playing functions take a table and any other required inputs, and generate a new table as the function result.

This example starts by asking how many players, besides the card dealer and the game user, should play a simulated Blackjack game. The game user controls when they want another card while the dealer and any other simulated players play automatically (they always hit when their card score is less than 17).

I define the types for playing cards and an entire card deck in the file *Card.hs*:

```
1  module Card (Card, Rank, Suit, orderedCardDeck, cardValue) where
2
3  import Data.Maybe (fromMaybe)
4  import Data.List (elemIndex)
5  import Data.Map (fromList, lookup, keys)
6
7  data Card = Card { rank :: Rank
8                   , suit :: Suit }
9                  deriving (Eq, Show)
10
11 data Suit = Hearts | Diamonds | Clubs | Spades
12           deriving (Eq, Show, Enum, Ord)
13
14 data Rank = Two | Three | Four
15           | Five | Six | Seven | Eight
16           | Nine | Ten | Jack  | Queen | King | Ace
17           deriving (Eq, Show, Enum, Ord)
18
```

```
rankMap = fromList [(Two,2), (Three,3), (Four,4), (Five,5),
                    (Six,6), (Seven,7), (Eight,8), (Nine,9),
                    (Ten,10), (Jack,10), (Queen,10),
                    (King,10), (Ace,11)]

orderedCardDeck :: [Card]
orderedCardDeck = [Card rank suit | rank <- keys rankMap,
                                    suit <- [Hearts .. Clubs]]

cardValue :: Card -> Int
cardValue aCard =
  case (Data.Map.lookup (rank aCard) rankMap) of
    Just n -> n
    Nothing -> 0 -- should never happen
```

As usual, the best way to understand this code is to go to the GHCi repl:

```
*Main Card RandomizedList Table> :l Card
[1 of 1] Compiling Card             ( Card.hs, interpreted )
Ok, modules loaded: Card.
*Card> :t orderedCardDeck
orderedCardDeck :: [Card]
*Card> orderedCardDeck
[Card {rank = Two, suit = Hearts},Card {rank = Two, suit = Diamonds},Card {rank \
= Two, suit = Clubs},Card {rank = Three, suit = Hearts},Card {rank = Three,
    ...
*Card> head orderedCardDeck
Card {rank = Two, suit = Hearts}
*Card> cardValue $ head orderedCardDeck
2
```

So, we have a sorted deck of cards and a utility function for returning the numerical value of a card (we always count ace cards as 11 points, deviating from standard Blackjack rules).

The next thing we need to get is randomly shuffled lists. The Haskell Wiki[35] has a good writeup on randomizing list elements and we are borrowing their function **randomizedList** (you can see the source code in the file *RandomizedList.hs*). Here is a sample use:

[35] https://wiki.haskell.org/Random_shuffle

```
*Card> :l RandomizedList.hs
[1 of 1] Compiling RandomizedList   ( RandomizedList.hs, interpreted )
Ok, modules loaded: RandomizedList.
*RandomizedList> import Card
*RandomizedList Card> randomizedList orderedCardDeck
[Card {rank = Queen, suit = Hearts},Card {rank = Six, suit = Diamonds},Card {ran\
k = Five, suit = Clubs},Card {rank = Five, suit = Diamonds},Card {rank = Seven, \
suit = Clubs},Card {rank = Three, suit = Hearts},Card {rank = Four, suit = Diamo\
nds},Card {rank = Ace, suit = Hearts},
  ...
```

Much of the complexity in this example is implemented in *Table.hs* which defines the type **Table** and several functions to deal and score hands of dealt cards:

- createNewTable :: Players -> Table. Players is the integer number of other players at the table.
- setPlayerBet :: Int -> Table -> Table. Given a new value to bet and a table, generate a new modified table.
- showTable :: Table -> [Char]. Given a table, generate a string describing the table (in a format useful for development)
- initialDeal :: [Card] -> Table -> Int -> Table. Given a randomized deck of cards, a table, and the number of other players, generate a new table.
- changeChipStack :: Int -> Int -> Table -> Table. Given a player index (index order: user, dealer, and other players), a new number of betting chips for the player, and a table, then generate a new modified table.
- setCardDeck :: [Card] -> Table -> Table. Given a randomized card deck and a table, generate a new table containing the new randomized card list; all other table data is unchanged.
- dealCards :: Table -> [Int] -> Table. Given a table and a list of player indices for players wanting another card, generate a new modified table.
- resetTable :: [Card] -> Table -> Int -> Table. Given a new randomized card deck, a table, and a new number of other players, generate a new table.
- scoreHands :: Table -> Table. Given a table, score all dealt hands and generate a new table with these scores. There is no table type score data, rather, we "score" by changing the number of chips all of the players (inclding the dealer) has.
- dealCardToUser :: Table -> Int -> Table. For the game user, always deal a card. For the dealer and other players, deal another card if their hand score is less than 17.
- handOver :: Table -> Bool. Determine is a hand is over.
- setPlayerPasses :: Table -> Table. Call this function when the payer passes. Other players and dealer are then played out automatically.

The implementation in the file *Table.hs* is fairly simple, with the exception of the use of Haskell lenses to access nested data in the table type. I will discuss the use of lenses after the program listing, but: as you are reading the code look out for variables starting with the underscore character _ that alerts the *Lens* system that it should create data accessors for these variables:

```haskell
{-# LANGUAGE TemplateHaskell #-}  -- for makeLens

module Table (Table (..), createNewTable, setPlayerBet, showTable, initialDeal,
              changeChipStack, setCardDeck, dealCards, resetTable, scoreHands,
              dealCardToUser, handOver, setPlayerPasses) where
  -- note: export dealCardToUser only required for ghci development

import Control.Lens

import Card
import Data.Bool
import Data.Maybe (fromMaybe)

data Table = Table { _numPlayers :: Int
                   , _chipStacks :: [Int] -- number of chips,
                                          -- indexed by player index
                   , _dealtCards :: [[Card]] -- dealt cards for user,
                                             -- dealer, and other players
                   , _currentPlayerBet :: Int
                   , _userPasses       :: Bool
                   , _cardDeck         :: [Card]
                   }
           deriving (Show)

type Players = Int

createNewTable :: Players -> Table
createNewTable n =
  Table n
        [500 | _ <- [1 .. n]] -- give each player (incuding dealer) 10 chips
        [[] | _ <- [0..n]] -- dealt cards for user and other players
                           -- (we don't track dealer's chips)
        20 -- currentPlayerBet number of betting chips
        False
        [] -- placeholder for random shuffled card deck

resetTable :: [Card] -> Table -> Int -> Table
resetTable cardDeck aTable numberOfPlayers =
  Table numberOfPlayers
        (_chipStacks aTable)       -- using Lens accessor
        [[] | _ <- [0..numberOfPlayers]]
        (_currentPlayerBet aTable) -- using Lens accessor
```

```
43          False
44          cardDeck
45
46      -- Use lens extensions for type Table:
47
48  makeLenses ''Table
49
50  showDealtCards :: [[Card]] -> String
51  showDealtCards dc =
52    (show [map cardValue hand | hand <- dc])
53
54  setCardDeck :: [Card] -> Table -> Table
55  setCardDeck newDeck =
56    over cardDeck (\_ -> newDeck)  -- change value to new card deck
57
58  dealCards :: Table -> [Int] -> Table
59  dealCards aTable playerIndices =
60    last $ scanl dealCardToUser aTable playerIndices
61
62  initialDeal cardDeck aTable numberOfPlayers =
63    dealCards
64      (dealCards (resetTable cardDeck aTable numberOfPlayers)
65                 [0 .. numberOfPlayers])
66      [0 .. numberOfPlayers]
67
68  showTable :: Table -> [Char]
69  showTable aTable =
70    "\nCurrent table data:\n" ++
71    "  Chipstacks: " ++
72    "\n    Player: " ++ (show (head (_chipStacks aTable))) ++
73    "\n    Other players: " ++ (show (tail (_chipStacks aTable))) ++
74    "\n  User cards: " ++ (show (head (_dealtCards aTable))) ++
75    "\n  Dealer cards: " ++ (show ((_dealtCards aTable) !! 1)) ++
76    "\n  Other player's cards: " ++ (show (tail (tail(_dealtCards aTable)))) ++
77    -- "\n  Dealt cards: " ++ (show (_dealtCards aTable)) ++
78    "\n  Dealt card values: " ++ (showDealtCards (_dealtCards aTable)) ++
79    "\n  Current player bet: " ++
80    (show (_currentPlayerBet aTable)) ++
81    "\n  Player pass: " ++
82    (show (_userPasses aTable)) ++ "\n"
83
84  clipScore aTable playerIndex =
```

```
85    let s = score aTable playerIndex in
86      if s < 22 then s else 0
87
88  scoreHands aTable =
89    let chipStacks2 = _chipStacks aTable
90        playerScore = clipScore aTable 0
91        dealerScore = clipScore aTable 1
92        otherScores = map (clipScore aTable) [2..]
93        newPlayerChipStack = if playerScore > dealerScore then
94                               (head chipStacks2) + (_currentPlayerBet aTable)
95                             else
96                               if playerScore < dealerScore then
97                                  (head chipStacks2) - (_currentPlayerBet aTable)
98                               else (head chipStacks2)
99        newOtherChipsStacks =
100         map (\(x,y) -> if x > dealerScore then
101                          y + 20
102                        else
103                          if x < dealerScore then
104                            y - 20
105                          else y)
106             (zip otherScores (tail chipStacks2))
107       newChipStacks  = newPlayerChipStack:newOtherChipsStacks
108   in
109     over chipStacks (\_ -> newChipStacks) aTable
110
111 setPlayerBet :: Int -> Table -> Table
112 setPlayerBet newBet =
113   over currentPlayerBet (\_ -> newBet)
114
115 setPlayerPasses :: Table -> Table
116 setPlayerPasses aTable =
117   let numPlayers = _numPlayers aTable
118       playerIndices = [1..numPlayers]
119       t1 = over userPasses (\_ -> True) aTable
120       t2 = dealCards t1 playerIndices
121       t3 = dealCards t2 playerIndices
122       t4 = dealCards t3 playerIndices
123   in
124     t4
125
126
```

```
127 changeChipStack :: Int -> Int -> Table -> Table
128 changeChipStack playerIndex newValue =
129   over chipStacks (\a -> a & element playerIndex .~ newValue)
130
131 scoreOLD aTable playerIndex =
132   let scores = map cardValue ((_dealtCards aTable) !! playerIndex)
133       totalScore = sum scores in
134     if totalScore < 22 then totalScore else 0
135
136 score aTable playerIndex =
137   let scores = map cardValue ((_dealtCards aTable) !! playerIndex)
138       totalScore = sum scores in
139     totalScore
140
141 dealCardToUser' :: Table -> Int -> Table
142 dealCardToUser' aTable playerIndex =
143   let nextCard = head $ _cardDeck aTable
144       playerCards = nextCard : ((_dealtCards aTable) !! playerIndex)
145       newTable = over cardDeck (\cd -> tail cd) aTable in
146     over dealtCards (\a -> a & element playerIndex .~ playerCards) newTable
147
148 dealCardToUser :: Table -> Int -> Table
149 dealCardToUser aTable playerIndex
150   | playerIndex == 0  = dealCardToUser' aTable playerIndex -- user
151   | otherwise         = if (score aTable playerIndex) < 17 then
152                             dealCardToUser' aTable playerIndex
153                         else aTable
154
155 handOver :: Table -> Bool
156 handOver aTable =
157   _userPasses aTable
```

In line 48 we use the function **makeLenses** to generate access functions for the type **Table**. We will look in some detail at lines 54-56 where we use the lense **over** function to modify a nested value in a table, returning a new table:

```
1 setCardDeck :: [Card] -> Table -> Table
2 setCardDeck newDeck =
3   over cardDeck (\_ -> newDeck)
```

The expression in line 3 evaluates to a partial function that takes another argument, a table, and returns a new table with the card deck modified. Function **over** expects a function as its second

argument. In this example, the inline function ignores the argument it is called with, which would be the old card deck value, and returns the new card deck value which is placed in the table value.

Using lenses can greatly simplify the code to manipulate complex types.

Another place where I am using lenses is in the definition of function **scoreHands** (lines 88-109). On line 109 we are using the **over** function to replace the old player betting chip counts with the new value we have just calculated:

```
  over chipStacks (\_ -> newChipStacks) aTable
```

Similarly, we use **over** in line 113 to change the current player bet. In function **handOver** on line 157, notice how I am using the generated function **_userPasses** to extract the value of the user passes boolean flag from a table.

The function **main**, defined in the file *Main.hs*, uses the code we have just seen to represent a table and modify a table, is fairly simple. A main game loop repetitively accepts game user imput, and calls the appropriate functions to modify the current table, producing a new table. Remember that the table data is immutable: we always generate a new table from the old table when we need to modify it.

```
1  module Main where
2
3  import Card   -- pure code
4  import Table  -- pure code
5  import RandomizedList  -- impure code
6
7  printTable :: Table -> IO ()
8  printTable aTable =
9    putStrLn $ showTable aTable
10
11 randomDeck =
12   randomizedList orderedCardDeck
13
14 gameLoop :: Table -> Int -> IO b
15 gameLoop aTable numberOfPlayers = do
16   printTable aTable
17   cardDeck <- randomDeck
18   if (handOver aTable) then
19     do
20       putStrLn "\nHand over. State of table at the end of the game:\n"
21       printTable aTable
22       putStrLn "\nNewly dealt hand:\n"
23       gameLoop (initialDeal cardDeck (scoreHands aTable)
```

```
                                    numberOfPlayers)
                                    numberOfPlayers
  else
    do
      putStrLn "Enter command:"
      putStrLn "  h)it or set bet to 10, 20, 30; any other key to stay:"
      command <- getLine
      if elem command ["10", "20", "30"] then
        gameLoop (setPlayerBet (read command) aTable) numberOfPlayers
      else
        if command == "h" then
          gameLoop (dealCards aTable [0 .. numberOfPlayers]) numberOfPlayers
        else
          gameLoop (setPlayerPasses (dealCards aTable [1 .. numberOfPlayers]))
                   numberOfPlayers
             -- player stays (no new cards)

main :: IO b
main = do
  putStrLn "Start a game of Blackjack. Besides yourself, how many other"
  putStrLn "players do you want at the table?"
  s <- getLine
  let num = (read s :: Int) + 1
  cardDeck <- randomDeck
  let aTable = initialDeal cardDeck (createNewTable num) num
  gameLoop aTable num
```

I encourage you to try playing the game yourself, but if you don't here is a sample game:

```
*Main Card RandomizedList Table> main
Start a game of Blackjack. Besides yourself, how many other
players do you want at the table?
1

Current table data:
  Chipstacks:
    Player: 500
    Other players: [500]
  User cards: [Card {rank = Three, suit = Clubs},Card {rank = Two, suit = Hearts\
}]
  Dealer cards: [Card {rank = Queen, suit = Diamonds},Card {rank = Seven, suit =\
 Clubs}]
```

```
  Other player's cards: [[Card {rank = King, suit = Hearts},Card {rank = Six, su\
it = Diamonds}]]
  Dealt card values: [[3,2],[10,7],[10,6]]
  Current player bet: 20
  Player pass: False

Enter command: h)it or set bet to 10, 20, 30; any other key to stay:
h

Current table data:
  Chipstacks:
    Player: 500
    Other players: [500]
  User cards: [Card {rank = Six, suit = Hearts},Card {rank = Three, suit = Clubs\
},Card {rank = Two, suit = Hearts}]
  Dealer cards: [Card {rank = Queen, suit = Diamonds},Card {rank = Seven, suit =\
 Clubs}]
  Other player's cards: [[Card {rank = Eight, suit = Hearts},Card {rank = King, \
suit = Hearts},Card {rank = Six, suit = Diamonds}]]
  Dealt card values: [[6,3,2],[10,7],[8,10,6]]
  Current player bet: 20
  Player pass: False

Enter command: h)it or set bet to 10, 20, 30; any other key to stay:
h

Current table data:
  Chipstacks:
    Player: 500
    Other players: [500]
  User cards: [Card {rank = King, suit = Clubs},Card {rank = Six, suit = Hearts}\
,Card {rank = Three, suit = Clubs},Card {rank = Two, suit = Hearts}]
  Dealer cards: [Card {rank = Queen, suit = Diamonds},Card {rank = Seven, suit =\
 Clubs}]
  Other player's cards: [[Card {rank = Eight, suit = Hearts},Card {rank = King, \
suit = Hearts},Card {rank = Six, suit = Diamonds}]]
  Dealt card values: [[10,6,3,2],[10,7],[8,10,6]]
  Current player bet: 20
  Player pass: False

Enter command: h)it or set bet to 10, 20, 30; any other key to stay:

```

```
Current table data:
  Chipstacks:
    Player: 500
    Other players: [500]
  User cards: [Card {rank = King, suit = Clubs},Card {rank = Six, suit = Hearts}\
,Card {rank = Three, suit = Clubs},Card {rank = Two, suit = Hearts}]
  Dealer cards: [Card {rank = Queen, suit = Diamonds},Card {rank = Seven, suit =\
 Clubs}]
  Other player's cards: [[Card {rank = Eight, suit = Hearts},Card {rank = King, \
suit = Hearts},Card {rank = Six, suit = Diamonds}]]
  Dealt card values: [[10,6,3,2],[10,7],[8,10,6]]
  Current player bet: 20
  Player pass: True

Hand over. State of table at the end of the game:

Current table data:
  Chipstacks:
    Player: 520
    Other players: [520]
  User cards: [Card {rank = King, suit = Clubs},Card {rank = Six, suit = Hearts}\
,Card {rank = Three, suit = Clubs},Card {rank = Two, suit = Hearts}]
  Dealer cards: [Card {rank = Queen, suit = Diamonds},Card {rank = Seven, suit =\
 Clubs}]
  Other player's cards: [[Card {rank = Eight, suit = Hearts},Card {rank = King, \
suit = Hearts},Card {rank = Six, suit = Diamonds}]]
  Dealt card values: [[10,6,3,2],[10,7],[8,10,6]]
  Current player bet: 20
  Player pass: True
```

Here the game user has four cards with values of [10,6,3,2] for a winning score of 21. The dealer has [10,7] for a score of 17 and the other player has [8,10,6], a value greater than 21 so the player went "bust."

I hope that you enjoyed this last example that demonstrates a reasonable approach for managing state when using immutable data.

Book Wrap Up

As I mentioned in the Preface, I had a slow start learning Haskell because I tried to learn too much at one time. In this book I have attempted to show you a subset of Haskell that is sufficient to write interesting programs - a gentle introduction.

Haskell beginners often dislike the large error listings from the compiler. The correct attitude is to recognize that these error messages are there to help you. That is easier said than donee, but try to be happy when the compiler points out an error - in the long run I find using Haskell's *fussy* compiler saves me time and lets me refactor code knowing that if I miss something in my refactoring the compiler will immediately let me know what needs to be fixed.

The other thing that I hope you learned working through this book is how effective repl based programming is. Most code I write, unless it is very trivial, starts its life in a GHCi repl. When you are working with somene else's Haskell code it is similarly useful to have their code loaded in a repl as you read.

I have been programming professionally for forty years and I use many programming languages. Once I worked my way through early difficulties using Haskell it has become a favorite programming language. I hope that you enjoy Haskell development as much as I do.

Appendix A - Haskell Tools Setup

I recommend that if you are new to Haskell that you at least do a minimal installation of *stack* and work through the first chapter using an interactive REPL. After experimenting with the REPL then do please come back to Appendix A and install support for the editor of your choice (or an IDE) and hlint.

stack

I assume that you have the Haskell package manager *stack* installed. If you have not installed *stack* yet please follow these directions[36].

After installing stack and running it you will have a directory ".stack" in your home directory where stack will keep compiled libraries and configuration data. You will want to create a file "~/.stack/stack.yaml" with contents similar to my stack configuration file:

```
templates:
  params:
      author-email: markw@markwatson.com
      author-name: Mark Watson
      category: dev
      copyright: Copyright 2016 Mark Watson. All rights reserved
      github-username: mark-watson
```

Replace my name and email address with yours. You might also want to install the package manager Cabal and the "lint" program hlint:

```
$ stack install cabal-install
$ stack install hlint
```

These installs might take a while so go outside for ten minutes and get some fresh air.

You should get in the habit of running hlint on your code and consider trying to remove all or at least most warnings. You can customize the types of warnings hlint shows: read the documentation for hlint[37].

[36] http://docs.haskellstack.org/en/stable/README.html
[37] https://github.com/ndmitchell/hlint#readme

Creating a New Stack Project

I have already created stack projects for the examples in this book. When you have worked through them, then please refer to the stack documentation for creating projects[38].

Emacs Setup

There are several good alternatives to using the Emacs editor:

- GEdit on Linux
- TextMate on OS X
- IntelliJ with the Haskell plugin (all platforms)

I use all three of these alternatives on occasion, but Emacs with *haskell-mode* is my favorite environment. There are instructions for adding *haskell-mode* to Emacs on the project home page on github[39]. If you follow these instructions you will have syntax hiliting and Emacs will understand Haskell indentation rules.

Do you want more of an IDE-like Development Environment?

I recommend and use the Intero Emacs package[40] to get auto completions and real time syntax error warnings. **Intero** is designed to work with *stack*.

I add the following to the bottom of my .emacs file:

(add-hook ‘haskell-mode-hook ‘intero-mode)

and if Intero is too “heavy weight” for my current project, then I comment out the add-hook expression. Intero can increase the startup time for Emacs for editing Haskell files. That said, I almost always keep Intero enabled in my Emacs environment.

hlint

hlint is a wonderful tool for refining your knowledge and use of the Haskell language. After writing new code and checking that it works, then run **hlint** for suggestions on how to improve your code.

Install **hlint** using:

[38] https://docs.haskellstack.org/en/stable/README/#start-your-new-project
[39] https://github.com/haskell/haskell-mode
[40] https://commercialhaskell.github.io/intero/

```
stack install hlint
```