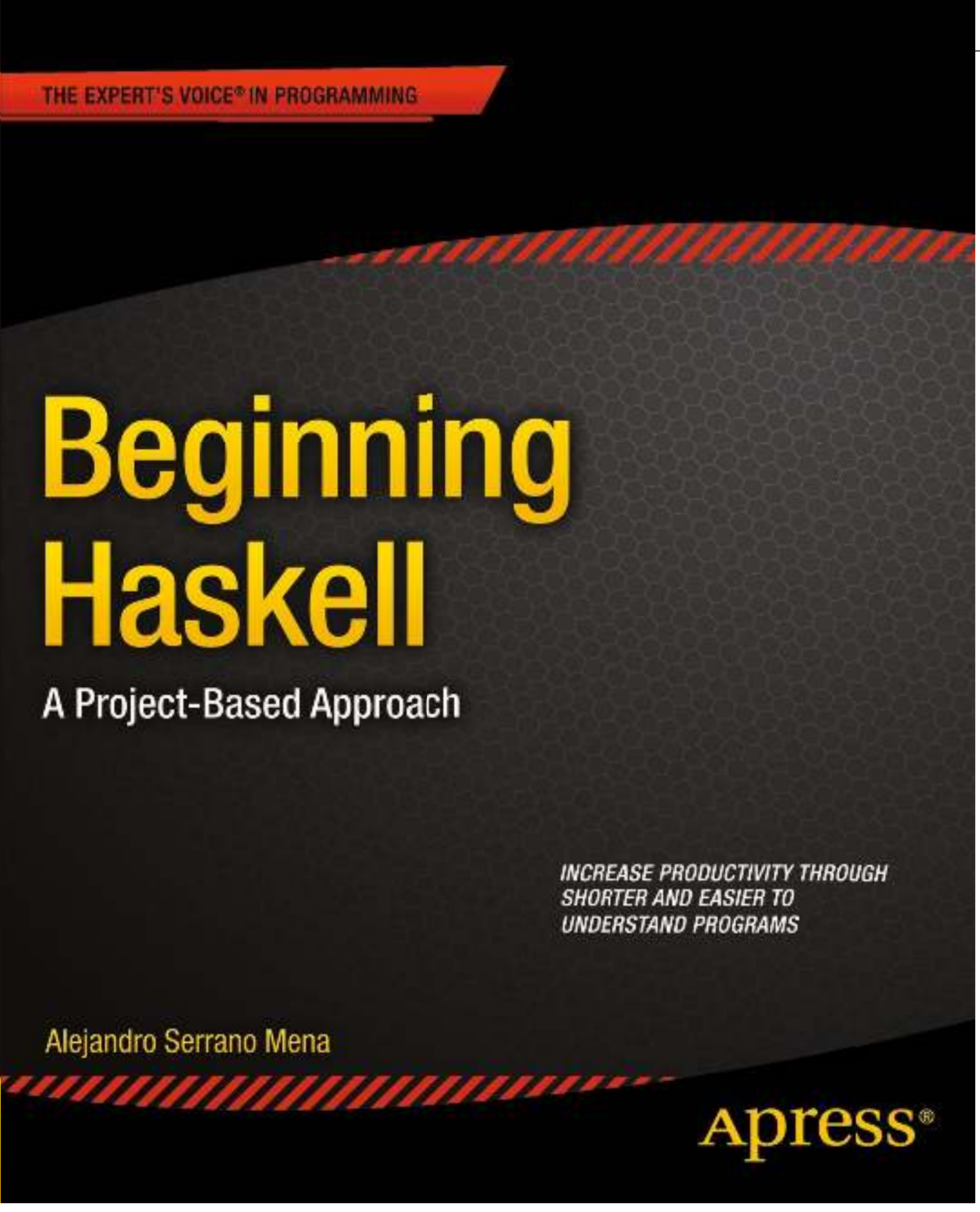
THE EXPERT'S VOICE® IN PROGRAMMING
Beginning Haskell
A Project-Based Approach
INCREASE PRODUCTIVITY THROUGH SHORTER AND EASIER TO UNDERSTAND PROGRAMS
Alejandro Serrano Mena
Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Functional programming is gathering momentum. Mainstream languages such as Java and C# are adopting features from this paradigm; and languages such as Haskell, Scala, Clojure, or OCaml, which embody functional programming from the very beginning, are being used in industry. Haskell is a noise-free, pure functional language with a long history, having a huge number of library contributors and an active community. This makes Haskell a great tool for both learning and applying functional programming.

Why You Should Learn Functional Programming

The rise in functional programming comes from two fronts. Nowadays, most applications are heavily *concurrent* or need to be *parallelized* to perform better. Think of any web server that needs to handle thousands of connections at the same time. The way you express the intent of your code using Haskell makes it easier to move from a single-thread application to a multi-threaded one at a negligible cost.

Apart from becoming more concurrent, applications are becoming much *larger*. You would like your development environment to help you catch bugs and ensure interoperation between all modules of your system. Haskell has a very strong type system, which means that you can express a wide range of invariants in your code, which are checked at compile time. Many of the bugs, which previously would be caught using tests, are now completely forbidden by the compiler. Refactoring becomes easier, as you can ensure that changes in your code do not affect those invariants.

Learning functional programming will put you in a much better position as a developer. Functional thinking will continue permeating through mainstream programming in the near future. You'll be prepared to develop larger and faster applications that bring satisfaction to your customers.

Why You Should Read this Book

This book focuses both on the *ideas* underlying and in the *practicalities* of Haskell programming. The chapters show you how to apply functional programming concepts in real-world scenarios. They also teach you about the tools and libraries that Haskell provides for each specific task. Newcomers to functional programming will not be the only ones who will benefit from reading this book. Developers of Scala, Clojure, Lisp, or ML will be also able to see what sets Haskell apart from other languages.

The book revolves around the project of building a web-based strorefront. In each of the five parts the focus is on a subsystem of this store: representing clients and products in-memory, data mining (including parallelization and concurrency), persistent storage, discount and offers, and the general architecture of the application. The topics have been carefully selected for you to get a glimpse of the whole Haskell ecosystem.

PART 1

First Steps

CHAPTER 1

Going Functional

Welcome to the world of Haskell! Before looking too deeply at the language itself, you will learn about what makes Haskell different from other languages and what benefits come with those differences. Haskell belongs to the family of *functional languages*, a broad set that includes ML, Lisp, Scala, and Clojure. If you have a background in imperative or object-oriented languages, such as C, C++, or Java, you will be introduced to the new ideas present in functional languages. If you already have experience with functional languages, you will see how other features in Haskell, such as lazy evaluation and type classes, make this language different from any other.

This book requires no previous experience with the functional paradigm. However, some general knowledge about programming (such as the concepts of loops and conditionals) and some minimal practice with the shell or console are assumed.

After introducing Haskell, I will review how to install Haskell on your system, including the EclipseFP development environment that brings Haskell and the Eclipse IDE together. Finally, you will take your first steps with the language in the Glasgow Haskell Compiler (GHC) interpreter, a powerful tool that executes expressions in an interactive way. Throughout the book you will develop parts of a time machine web store; as with many things in life, the best way to learn Haskell is by writing Haskell programs!

Why Haskell?

If you are reading this book, it means you are interested in learning Haskell. But what makes this language special? Its approach to programming can be summarized as follows:

- Haskell belongs to the family of *functional* languages.
- It embodies in its core the concept of *purity*, separating the code with side effects from the rest of the application.
- The evaluation model is based on *laziness*.
- *Types* are *statically checked* by the compiler. Also, Haskell features a type system that is much *stronger* and expressive than usual.
- Its approach to polymorphism is based on *parametricity* (similar to generics in Java and C#) and *type classes*.

In the rest of this section, you will understand what the terms in this list mean and their implications when using Haskell. Also, you will get a broad view of the entire Haskell ecosystem in a typical distribution: the compiler, the libraries, and the available tools.

Why Pure Functional Programming?

Functional programming is one of the styles, or *paradigms*, of programming. A programming paradigm is a set of concepts shared by different programming languages. For example, Pascal and C are part of the imperative paradigm, and Java and C++ mix the imperative paradigm with the object-oriented one. The fundamental emphasis of functional programming is the empowerment of *functions* as *first-class citizens*. This means functions can be manipulated like any other type of data in a program. A function can be passed as an argument to another function, returned as a result, or assigned to a variable. This ability to treat functions as data allows a higher level of abstraction and therefore more opportunities for reuse.

For example, consider the task of iterating through a data structure, performing some action on each element. In an object-oriented language, the implementer of the structure would have surely followed the iterator pattern, and you as a consumer would write code similar to the following Java code:

```
Iterator it = listOfThings.iterator();
while (it.hasNext()) {
    Element e = it.next();
    action(e); // perform the action
}
```

As you can see, there is a lot of boilerplate code in the example. In Haskell, you would use the `map` function, which takes as its argument the action to perform on each element. The corresponding code is as follows:

```
map listOfThings action
```

The code now is much more concise, and the actual intention of the programmer is explicit from the use of the `map` function. Furthermore, you prevent any possible issue related to applying the iterator pattern poorly because all the details have been abstracted in a function. Actually, a function such as `map` is common in functional code, which gives you confidence that any bug in its implementation will be found quickly.

Performing the same task in Java (up to version 7) requires, on the provider side, you to create an interface that contains the function that will perform the operation. Then, on the user side, you need to implement that interface through a class or use an anonymous class. This code will be much longer than the one-line version you saw earlier. In fact, new versions of Java (from version 8 on), C++, and C# (from the release of .NET Framework 3.5) are embracing functional concepts and will allow you to write code similar to the previous line.

In Haskell, a piece of code consists of *expressions*, which are evaluated in a similar fashion to mathematical expressions. In an imperative language, methods consist of statements that change a global state. This is an important distinction because in an imperative program the same piece of code may have different results depending on the initial state when it is executed. It's important to notice here that elements outside of the program control (known as *side effects*), such as input and output, network communication, and randomness, are also part of this global state that may change between executions of the same function.

Expressions in Haskell cannot have side effects by default; these expressions are called *pure*. A common misunderstanding about functional programming is that it disallows any kind of change to the outer state. This is not true; side effects are possible in Haskell, but the language forces the programmer to separate the pure, side-effect-free parts from the "impure" ones.

The main benefits of purity are the improved ability to reason about the code and an easier approach for testing the code. You can be sure that the outcome of a pure function depends only on its parameters and that every run with the same inputs will give the same result. This property is called *referential transparency*, and it's the foundation for applying formal verification techniques, as you will see in Chapter 15.

Pure functions are easier to compose because no interference comes to life in their execution. Actually, the evaluation of pure expressions is not dependent on the order in which it is done, so it opens the door to different *execution strategies* for the same piece of code. This is taken advantage of by the Haskell libraries providing parallel and concurrent execution and has even been used for scheduling code in a GPU in the Accelerate library.

By default, Haskell uses an execution strategy called *lazy evaluation*. Under laziness, an expression is never evaluated until it is needed for the evaluation of a larger one. Once it has been evaluated, the result is saved for further computation, or it's discarded if it's not needed in any other running code. This has an obvious benefit because only the minimal amount of computation is performed during program execution, but it also has drawbacks because all the suspended expressions that have not yet been evaluated must be saved in memory. Lazy evaluation is powerful but can become tricky, as you will see in Chapter 5.

Why Strong Static Typing?

Type systems come in various formats in almost all programming languages. A *type system* is an abstraction that categorizes the values that could appear during execution, tagging them with a so-called type. These types are normally used to restrict the possible set of actions that could be applied to a value. For example, it may allow concatenating two strings but forbid using the division operator between them.

This tagging can be checked, broadly speaking, at two times: at execution time (*dynamic* typing), which usually comes in languages with looser typing and allows implicit conversions between things such as integers and string, or at the moment of compilation (*static* typing), in which case programs must be validated to be completely well typed in terms of the language rules before generating the target output code (usually machine code or bytecode) and being allowed to run. Haskell falls into this second category: all your programs will be type checked before they are executed. Within statically typed languages, some of them, such as Java or C#, need to perform extra type checking at runtime. In contrast, once a Haskell program has been compiled, no more type checks have to be done, so performance is vastly increased.

Haskell's type system is very *strong*. Strength here means the number of invariants that can be caught at compile time before an error materializes while the application is running. This increases the confidence in code that is type checked, and it's common to hear the following in Haskell circles: "Once it compiles, it works." This strong typing gives rise to a way of programming dubbed *type-oriented programming*. Basically, programmers know the type of the function they are developing and have a broad idea of the structure of the code. Then, they "fill in the holes" with expressions from the surrounding environment that fit into it. This approach has actually been formalized, and there is another language similar to Haskell, called Agda,[1] which comes with an interactive programming environment that helps in filling in the holes and even does so automatically if only one option is available at one place.

In Chapters 13 and 15, I will move a bit from Haskell to Idris,[2] a language with a similar syntax that features *dependent typing*. Dependent typing is an even stronger form of type checking, where you can actually express invariants such as "If I concatenate a list of `n` elements to a list with `m` elements, I get back a list with `n+m` elements" or "I cannot get the first element of an empty list." Then, you will see how some of these techniques can be transferred as patterns into Haskell.

The last difference in Haskell with respect to typing comes from polymorphism. The problem is twofold. First, you want to write functions on lists without caring about the type of the elements contained in them. This is known as *parametric polymorphism*, and you will explore it in Chapter 3. In other cases, you want to express the fact that some types allow some specific operations on their values. For example, the idea of applying a function to all elements in a list, as you did before with `map`, can be generalized into the concept of having an operation that applies a function to all elements in some data structure, such as a tree or a graph. The solution here is called *type classes*, which group different types with a common interface. You will look at it in Chapter 4, where you will also realize that this concept is a very high-level one that allows for expressing several abstract ideas (functors, monads) and that gives an interesting flavor to Haskell code.

[1] Agda 2 web site: http://wiki.portal.chalmers.se/agda
[2] Idris web site: www.idris-lang.org/

The Haskell Ecosystem

Until now I have spoken only about Haskell the language. But the benefits of Haskell come not only from the language but also from the large and growing set of tools and libraries that can be used with the language.

Several compilers for Haskell are available, which usually take the name of a city: GHC[3] (from Glasgow), UHC[4] (from Utrecht), and JHC[5] (from the United States) are the maintained ones at the time of this writing. Of those, GHC is usually taken as the standard, and it's the one with the biggest number of features. You will follow this path and will work with GHC throughout the book.

Like any other popular programming language, Haskell has an online repository of libraries. It is called Hackage, and it's available at `http://hackage.haskell.org/`. Hackage integrates seamlessly with Cabal, the building tool for Haskell projects. In Hackage you can find libraries ranging from bioinformatics to game programming, window managers, and much more.

Apart from GHC and Cabal, in the book you will look at some tools that aim to help developers write better code faster. The first one will be the GHC profiler; you will learn about it in Chapter 5 to detect space and time leaks. You will also look at Hoogle and Haddock, which are used to browse and create documentation. In Chapter 14, you will use the UU Attribute Grammar System to help you build domain-specific languages.

The History of Haskell

Haskell is usually considered the successor of the Miranda programming language, which was one of the most important lazy functional programming languages in the 1980s. However, at that time, lots of other languages of the same kind existed in the wild. That made it difficult for researchers to have a common base in which to perform experiments in this paradigm. So, by the end of that decade, they decided to build a completely new language that would become the groundwork for that research.

During the 1990s, several versions of the language were produced. During this time Haskell evolved and incorporated some of the features that give the language its particular taste, such as type classes and monads for managing input and output. In 1998, a report defined Haskell 98, which was taken as the standard for any compliant Haskell compiler. This is the version targeted by most library developers.

However, new ideas from researchers and compiler writers were integrated into Haskell compilers, mostly in GHC. Some of these extensions became widely used, which made the case for a revised Haskell standard, which came out in 2010. At the time of this writing, GHC targets this version of the language.

As the language has become more popular, more extensions have been added to GHC and other compilers, and these features usually can be switched on or off at the developer's will. As a result, a more disciplined schedule has been created for issuing revised Haskell standards on a timely basis.

Your Working Environment

At this point you are probably feeling the need to try Haskell on your own computer. The first step for this is, of course, to have a working Haskell installation on your system. Haskell developers worried in the past about how to get people ready fast and easily. So, they created the Haskell Platform, a distribution containing the GHC compiler, the Cabal build and library system, and a comprehensive set of libraries. To get the Haskell Platform, go to `www.haskell.org/platform/`. Then, follow the steps corresponding to the operating system you will be using.

[3]GHC web site: `www.haskell.org/ghc/`
[4]UHC web site: `www.cs.uu.nl/wiki/UHC`
[5]JHC web site: `http://repetae.net/computer/jhc/`

Installing on Windows

Installing on the Microsoft operating system is easy because the file you download is an executable that will take care of everything.

Installing on Mac OS X

To develop with Haskell on the Apple operating system, you need Xcode as a prerequisite, which contains the basic developer tools for the environment. The Xcode component is free, but Apple asks you to register before you can download it. The website to visit is `https://developer.apple.com/xcode/`.

Installing Apple's Xcode package will give you the graphical tools, but what you need are the console tools. To get them, open the Xcode IDE and go to the Preferences window. There, click the Downloads tab. You will see a Command Line Tools item, which is the one you should install. See Figure 1-1, which shows this tool highlighted and also the Install button that you must click to accomplish the install.

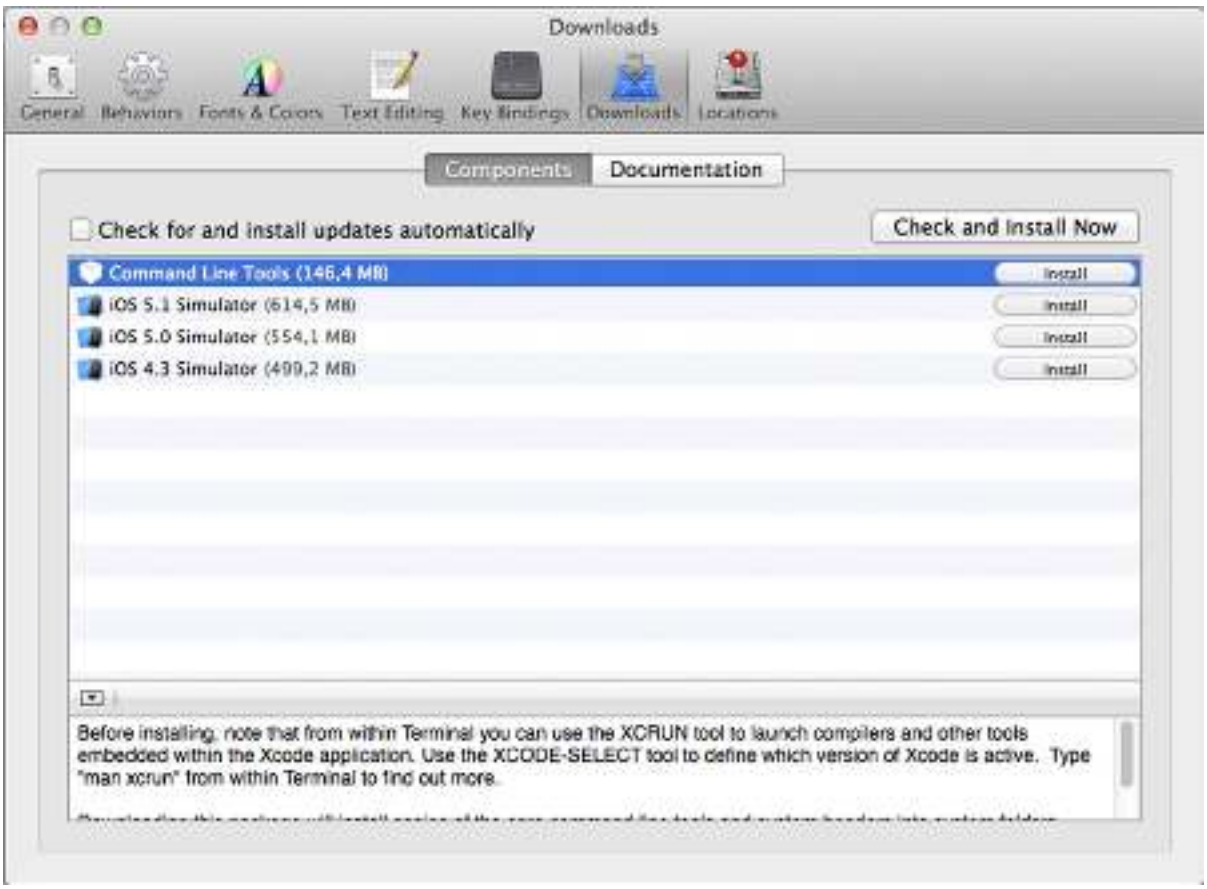

Figure 1-1. *Installing the XCode Command Line Tools*

The next decision you need to make regarding the Haskell Platform installer is whether you need the 32- or 64-bit version. There is no important difference between them. However, if you plan to interface some external libraries with the Haskell compiler, be sure to download the version that matches the architecture those libraries were compiled in.

Installing on Linux

The world of Linux distributions is diverse, so it's difficult to suggest the best way to get a working Haskell installation on Linux systems. If you use a distribution supporting some sort of package management system, it's better to stick with that system. For example, Debian-based systems, like Ubuntu, support `apt-get`. Thus, you can run the following[6]:

```
$ sudo apt-get install haskell-platform
```

In Fedora and other Red Hat–based distros, this is the line to run:

```
$ yum install haskell-platform
```

You can also check the whole list of distributions that have Haskell Platform out of the box on the Haskell Platform website.

Installing on Linux from Source

In case you want or need to perform a complete installation from source code, you must follow these steps:

1. Go to the GHC compiler web page, www.haskell.org/ghc/. Click the Download link and get the binary package for the latest stable release.
2. Uncompress the file you just downloaded into the place you want it to live. It's recommended that you add that folder to your `PATH`. You may need to install some libraries, like GMP, to be able to run this binary.

■ **Note** You can also build GHC from source. However, this is a tedious and error-prone process, so using just the binary distribution is recommended. In case you want to follow that path, the Haskell wiki page has a detailed description of the process; see http://hackage.haskell.org/trac/ghc/wiki/Building.

---

3. Return to the Haskell Platform page to download its source.
4. Uncompress, build, and install it, which is usually accomplished by running this:

```
$ tar -xzvf haskell-platform-*.tar.gz
$ cd haskell-platform-*
$ ./configure
$ make
$ make install
```

[6]In this book, anything to be input in the console will be preceded with the $ sign.

Checking That the Installation Is Successful

It's now time to see whether your Haskell Platform is correctly installed. To do so, open a console, type `ghci -e 5+3`, and press Enter. You should see `8` as output.

Installing EclipseFP

One of the most powerful environments for programming in Haskell is the one integrated in the Eclipse IDE, called EclipseFP. It provides syntax highlighting, code autocompletion, integrated documentation, profiling and unit testing, and graphical wizards for managing Cabal projects and creating new applications and modules. Additionally, EclipseFP provides an easy bridge for those developers coming from other environments such as Java or .NET, where the use of integrated programming environments is the norm. Being integrated in Eclipse, it benefits from the great range of plug-ins available, which includes Subversion and Git support. During the book, you will learn in parallel how to use the command line and EclipseFP to manage Haskell projects.

■ **Note** Almost all text editors targeted to developers have some mode or plug-in for Haskell. Emacs and Vim have powerful modes with features such as autocompletion. Others such as Sublime Text, Notepad++, Kate, and Gedit are mostly limited to syntax highlighting.

To get EclipseFP working, you must first get Eclipse. (EclipseFP runs in any release of Eclipse from 3.7 onward.) Go to `http://eclipse.org/downloads`. As you can see, there are several options, which depend on the language support that is bundled by default with each package. In this case, the choice doesn't matter because you will be adding Haskell support by hand. After downloading, just uncompress the file and execute the Eclipse binary. On Mac OS X, you may want to move the entire folder to Applications. In case it doesn't work out of the box, check whether your system has a Java Virtual Machine available, version 6 or newer.

Eclipse uses the concept of workspaces. A *workspace* is a set of related projects in which you work at the same time and which share preferences such as active plug-ins or location of the windows. The first time you open the IDE, it will ask you where to create your workspace; pick a spot on your system that's easy to find. I recommend creating a new workspace for working through the examples in this book.

The next step is getting the EclipseFP plug-in. For that, click the Eclipse Help menu, and select the Install New Software option. In the Work with field, enter `http://eclipsefp.sf.net/updates` and press Enter. After some seconds, a list containing "Functional programming" should appear. Select the FP: Haskell Support for Eclipse option shown in Figure 1-2, and click Next a couple of times. Eclipse will start downloading this new feature and integrating it with the environment.

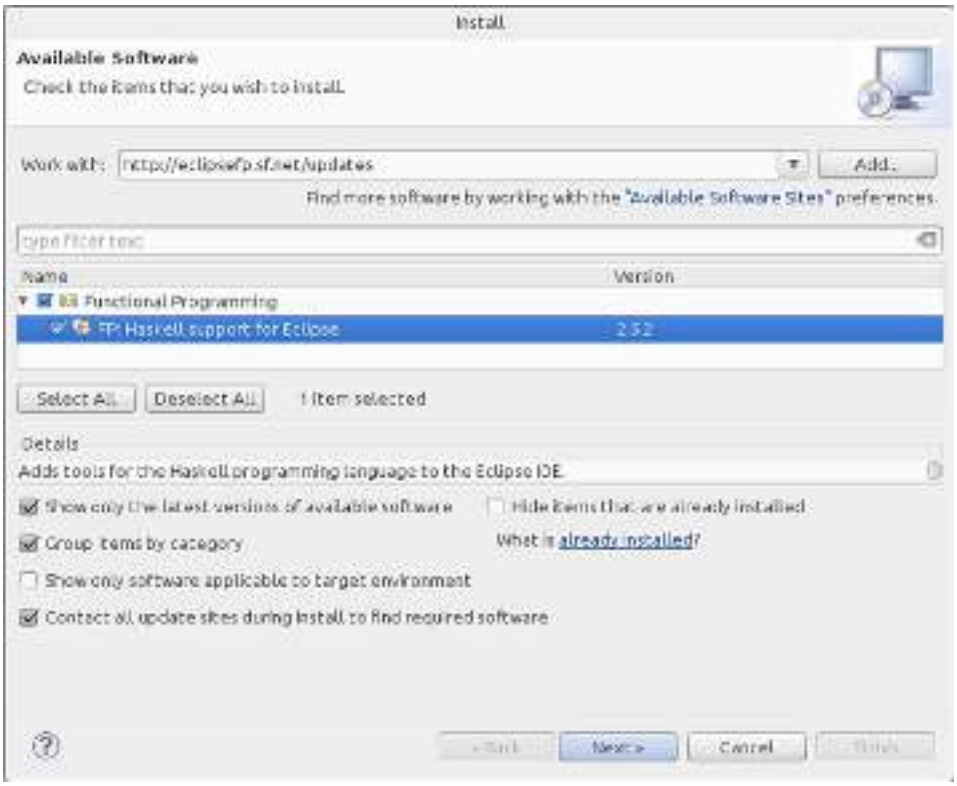

Figure 1-2. *Eclipse plug-in installation window*

After installation is finished, you will be asked to restart Eclipse. Click OK to agree. Once the IDE starts, EclipseFP will ask you about installing helper executables; you'll receive the dialog shown in Figure 1-3. These are programs that are not needed by the core of Haskell editing but greatly enhance the experience of the plug-in. Check the boxes to install them. Depending on your machine, this process may take a long time because several libraries and programs must be downloaded and compiled. You can check the progress on the Console tab.

Figure 1-3. *EclipseFP helper executables*

The final step is to go into the Haskell mode. In the top-right corner, you'll see a small Open Perspective button. Once you click it, a list of the supported environments for Eclipse appears. Click Haskell. This first time running EclipseFP, you will be prompted to download information from autocompletion from the Internet. It's recommended you do so in order to get better feedback messages.

Taking Your First Steps with GHCi

Following the installation of Haskell, you verified the installation's success by running GHCi. This application is one instance of a *read-eval-print loop* (REPL), or, more succinctly, an *interpreter*. In GHCi, you input an expression and press Enter. The expression gets evaluated, and the result is shown on the screen. This allows for a programming methodology where you navigate into and discover the functionality of a library by issuing commands in the interpreter and also test your code interactively.

To open an interpreter in a console, just run `ghci`. A prompt with `Prelude>` at the beginning should appear. This line tells you that the interpreter is ready to accept commands and that the only loaded module at this moment is the Prelude, which contains the most basic functions and data types in Haskell. As a first approximation, GHCi can work as a fancy calculator, as shown here:

```
Prelude> 5 * 3
15
Prelude> 1/2 + 1/3
0.8333333333333333
```

If you now type `s` and press the Tab key, you will see a list of all possible functions beginning with that letter. If you then type `q` and press Tab again, only one possibility is left, `sqrt`, which is automatically written for you. One distinguishing choice made by Haskell creators was that parentheses are not used when applying a function. This means that if you want to find the square root of 7, you just write this:

```
Prelude> sqrt 7
2.6457513110645907
```

There are many other arithmetic operations you can perform in the interpreter: `sin`, `cos`, `log`, `exp`, and so forth. In the next chapter you will learn how to use strings and lists and how to define functions, which will make your experience with the interpreter much more rewarding.

GHCi does not by default allow you to input several lines of code. For example, if you want to break the previous addition of two rational numbers into two lines, you cannot do it easily. Try entering the expression again, but press Enter after inputting the plus sign. If you press Enter, this error message will be produced:

```
Prelude> 1/2 +
<interactive>:2:6:
    parse error (possibly incorrect indentation or mismatched brackets)
```

The solution is to start a *multiline block*. A multiline block is an expression that is allowed to span more than one line. To do so, enter `:{` and then press Enter. The prompt will change into `Prelude|`, showing that the input is expected to fill several lines. To end the block, enter the opposite of the beginning symbol, `:}`. Here's an example:

```
Prelude> :{
Prelude| 1/2 +
Prelude| 1/3
Prelude| :}
0.8333333333333333
```

■ **Caution** To start a multiline block, `:{` must be the only text entered in the first line.

All the internal actions of the interpreter (that is, those that are not functions on any library) start with a colon. For example, typing `:?` and pressing Enter lists all the available commands. Other possibilities are looking at the language standard version you are using, in this case Haskell 2010 with some customizations. Here's an example:

```
Prelude> :show language
base language is: Haskell2010
with the following modifiers:
  -XNoDatatypeContexts
  -XNondecreasingIndentation
```

I stated before that Haskell has a strong static type system. You can check that it forbids dividing two strings (which are written between double quotes), producing an error when input in the interpreter, like so:

```
Prelude> "hello" / "world"
<interactive>:2:9:
    No instance for (Fractional [Char]) arising from a use of `/'
    Possible fix: add an instance declaration for (Fractional [Char])
    In the expression: "hello" / "world"
    In an equation for `it': it = "hello" / "world"
```

`Fractional` is the name of the type class that provides support for the `/` operator. The error message is saying that in order to be able to divide two strings, you should tell the compiler how to do so, by adding a declaration with the code for the `Fractional` type class in the case of strings.

To close the interpreter and go back to the console, you can issue the command `:quit` or just press the key combination Ctrl+D. In both cases the result is the same.

```
Prelude> :quit
Leaving GHCi.
```

■ **Note** GHCi is a powerful and customizable tool. You can find lots of tips and tricks on the Haskell wiki page devoted to the interpreter, www.haskell.org/haskellwiki/GHC/GHCi.

The Time Machine Store

If you have already taken a look at the table of contents of this book, you will have noticed that it is divided in four parts. Each part is devoted to a different module of a small web store.

> In this first part, you will learn how to define the basic blocks of your application, representing clients, products, and orders, and how to manage them in-memory.
>
> In Part 2, you will develop some data mining algorithms to get a better knowledge of the clients. In particular, you will develop a classification algorithm based on K-means and a recommendation algorithm.

Part 3 will deal with saving data into a persistent store. For product data you will use a custom file format, and for clients and orders you will use a more traditional database solution. With all of this, you will be able to build the initial application by Chapter 12.

Finally, in Part 4 you will see how a domain-specific language can be used to model special offers in the system, such as "20 percent discount for all clients in Europe younger than 30."

What will you sell in this store? Time machines!

> *Welcome to the exciting world of time machines! These machines are quite special, and our clients come from all parts of the universe to get one. We would like to have a web store to handle all the orders. And we would also like to be developed in a language as special as our machines, like Haskell.*

Sound exciting? Throughout this book you'll be using Haskell to build your very own store for selling time machines. It's a fun example, and it should keep the book interesting.

Summary

In this chapter you got familiar with Haskell.

- You learned about the distinguishing features of pure functional programming and how it helps to build code that is more concise, more maintainable, and less error prone.
- You looked at the benefits of having a strong, statically checked type system, like the one embodied in Haskell, and how dependent typing makes it possible to express invariants in a powerful way.
- The major tools in the Haskell ecosystem were introduced: the GHC compiler, the Cabal build tool, the Hackage library repository, and the GHC interpreter. You also took your first steps with the interpreter.
- You looked at the installation process of the Haskell Platform in the most common computer environments, as well as the Haskell plug-in for the Eclipse integrated development environment.
- You were presented with the main target in the book (apart from learning Haskell): building a web store focused on selling time machines, with modules for simple data mining and offer descriptions.

CHAPTER 2

Declaring the Data Model

You already know how to get a working installation of the Haskell Platform and of the EclipseFP development environment. The next step toward your Time Machine Store is to create the initial set of values and functions that will represent the data in the system: clients, machines, and orders.

This chapter will give you the basic ingredients for creating these values and functions. In a first approximation, you will create functions operating on basic types. You already know numbers, and you will add lists and tuples to the mix. Afterward, you will see how to create your own *algebraic data types* (ADTs) to better represent the kind of values you are interested in here. As part of this, you will learn about *pattern matching* for the first time, a powerful idiom to write concise code that follows closely the shape of the types.

Sometimes ADTs and pattern matching lead to code that's not clear enough. *Records* introduce some syntactic forms that make values easier to create and modify, and they are a well-known tool of Haskell programmers. In addition, you will look at two design patterns that are common in Haskell libraries, namely, *smart constructors* and *default values*.

This chapter will also introduce how to manage projects using Cabal and EclipseFP. In particular, you will see how to create a new project using both systems, along with the usual structure in folders, and how to load the code into the GHC interpreter to interactively test it.

Working with Characters, Numbers, and Lists

Characters and numbers are universally accepted as the most basic kind of values that a language should provide to programmers. Haskell follows this tradition and offers dedicated character and number types that will be introduced in this section. Afterward, you will see how to put together several of these values to create strings or lists of numbers, as well as the basic operations you can perform on any kind of list.

Characters

In some programming languages, numbers are also used to represent characters, usually in some encoding such as ASCII or Unicode. But following its tradition of clearly separating different concerns of a value, Haskell has a special type called `Char` for representing character data. To prevent problems with locales and languages, a `Char` value contains one Unicode character. These values can be created in two ways.

- Writing the character itself between single quotes, like `'a'`.
- Writing the code point, that is, the numeric value which represents the character as defined in the Unicode standard, in decimal between `'\` and `'` or in hexadecimal between `'\x` and `'`. For example, the same `'a'` character can be written as `'\97'` or `'\x61'`.

Using GHCi, you can check the actual type of each expression you introduce in the system. To do so, you use the `:t` command, followed by the expression. Let's check that characters indeed are characters.

```
Prelude Data.Char> :t 'a'
'a' :: Char
```

Let's now explore some of the functionality that Haskell provides for `Char`s. Only a few functions are loaded by default, so let's import a module with a lot more functions, in this case `Data.Char`.

```
Prelude> import Data.Char
Prelude Data.Char>
```

The prompt of the interpreter changes to reflect the fact that now two different modules are loaded. Furthermore, if you now write `to` and press Tab, you will see a greater number of functions than before. In Haskell, everything has its own type, so let's try to find out `toUpper`'s type.

```
Prelude Data.Char> :t toUpper
toUpper :: Char -> Char
```

The *arrow syntax* (shown as `->`) is used to specify types of functions. In this case, `toUpper` is a function taking a character (the `Char` on the left side) and returning another one (because of the `Char` on the right side). Of course, types don't have to be equal. For example, `chr` takes an integer and gives the character with that code point.

```
Prelude Data.Char> chr 97
'a'
Prelude Data.Char> :t chr
chr :: Int -> Char
```

For functions with more than one parameter, each argument type is separated from the next with a single arrow. For example, if you had a `min` function taking two integers and returning the smallest one, the type would be as follows:

```
min :: Integer -> Integer -> Integer
```

I mentioned in the previous chapter that Haskell is very strict at checking types. You can indeed verify this: if you try to apply the `chr` function to a character, the interpreter refuses to continue.

```
Prelude Data.Char> chr 'a'
<interactive>:7:5:
    Couldn't match expected type `Int' with actual type `Char'
    In the first argument of `chr', namely 'a'
    In the expression: chr 'a'
    In an equation for `it': it = chr 'a'
```

Numbers

In Chapter 1 you may have noticed that several kinds of numeric constants were used. Like most programming languages, Haskell supports a great variety of number types, depending on the width, precision, and support for decimal parts.

- `Int` is the bounded integer type. It supports values between at least ±536870911 (even though GHC uses a much wider range). Usually, values of the `Int` type have the native width of the architecture, which makes them the fastest.
- `Integer` is an unbounded integer type. It can represent any value without a decimal part without underflow or overflow. This property makes it useful for writing code without caring about bounds, but it comes at the price of speed.
- The Haskell base library also bundles exact rational numbers using the `Ratio` type. Rational values are created using `n % m`.
- `Float` and `Double` are floating-point types of single and double precision, respectively.

Haskell is strict with the types. If you need to convert between different numeric representations, the functions `fromInteger`, `toInteger`, `fromRational`, and `toRational` will help you deal with conversions. For example, you can switch between rational and floating-point representations of values. The `toRational` function tries to create a `Ratio` not far from the original value (this depends on its width), and you can move from rational to floating-point by dividing the numerator by the denominator of the ratio. Be aware that many of these functions are found in the `Data.Ratio` module, so you should import it first.

```
Prelude> import Data.Ratio
Prelude Data.Ratio> 1 % 2 + 1 % 3
5 % 6
Prelude Data.Ratio> toRational 1.3
5854679515581645 % 4503599627370496
Prelude Data.Ratio> toRational (fromRational (13 % 10))
5854679515581645 % 4503599627370496
```

As you can see from the examples, perfect round-tripping between rational and floating-point values is not always possible.

If you try to find the type of numeric constants, you may get a puzzling result.

```
Prelude> :t 5
5 :: Num a => a
Prelude> :t 3.4
3.4 :: Fractional a => a
```

Instead of making a numeric constant of a specific type, Haskell has a clever solution for supporting constants for different types: they are called *polymorphic*. For example, `5` is a constant that can be used for creating values of every type supporting the `Num` type class (which includes all types introduced before). On the other hand, `3.4` can be used for creating values of any type that is `Fractional` (which includes `Float` and `Double` but not `Int` or `Integer`). You will read in detail about type classes in Chapter 4, but right now you can think of a type class as a way to group sets of types that support the same operations.

■ **Caution** Since Haskell doesn't use parentheses in function invocations, that is, you write `f a b` instead of `f(a,b)`, you must be a bit more careful than usual when using negative numbers. For example, if you write `atan -4` in GHCi, you will get an error indicating `No instance for (Num (a0 -> a0)) arising from a use of '-'`. This means it has interpreted that you are trying to compute the subtraction of `atan` and `4`. To get the arctangent of `-4`, you should instead write `atan (-4)`.

Strings

After playing for some time with characters, you may wonder whether you can have a bunch of them together, forming what is commonly known as a *string*. The syntax for strings in Haskell is similar to C: you wrap letters in double quotes. The following code creates a string. If you ask the interpreter its type, what do you expect to get back?

```
Prelude Data.Char> :t "Hello world!"
"Hello world!" :: [Char]
```

Instead of some new type, like `String`, you see your old friend `Char` but wrapped in square brackets. Those brackets indicate that `"Hello world!"` is not a character but a *list* of characters. In general, given a type τ, the notation $[\tau]$ refers to the type of all lists whose elements are all of type τ. Lists are the most used data structure in functional programming. The fact that a type like a list depends on other types is known as *parametric polymorphism*, and you will delve into the details of it in the next chapter. Right now, let's focus on the practical side.

Lists

List *literals* (that is, lists whose values are explicitly set into the program code) are written with commas separating each of the elements, while wrapping everything between square brackets. As I have said, there's also special string syntax for a list of characters. Let's look at the types of some of these literals and the functions `reverse`, which gives a list in reverse order, and `(++)`, which concatenates two lists.

```
Prelude> :t [1,2,3]
[1, 2, 3] :: Num t => [t]
Prelude> :t reverse
reverse :: [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> reverse [1,2,3]
[3,2,1]
Prelude> reverse "abc"
"cba"
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Notice from this example that there are functions, such as `reverse` and `(++)`, that are able to operate on any kind of list. This means once you know them, you can apply your knowledge of them to any list (including strings of characters). To tell this fact, these functions show in its type a *type variable*. It is a variable because it can be replaced by any type because regular variables can take different values. Type variables must be written in code starting with lowercase letters, and they consist usually of one or two letters. Here, the type variable is shown as `a`.

■ **Note** Functions whose names are built entirely by symbols, like ++, must be called using infix syntax, writing them between the arguments instead of in front of them. So, you write `a ++ b`, not `++ a b`. In the case where you want to use the function in the normal fashion, you must use parentheses. So, you can write `(++) a b`.

Lists in Haskell are *homogeneous*: each list can handle elements of only a single type. Because of that, you are forbidden to create a list containing integers and characters and also to concatenate two lists with different kinds of elements.

```
Prelude> [1,2,3,'a','b','c']
<interactive>:13:2:
    No instance for (Num Char) arising from the literal '1'
    Possible fix: add an instance declaration for (Num Char)
Prelude> "abc" ++ [1,2,3]
<interactive>:11:11:
    No instance for (Num Char) arising from the literal '1'
    Possible fix: add an instance declaration for (Num Char)
```

List Operations

Like in most functional languages, lists in Haskell are linked lists. Such lists are composed of a series of cells that hold the values in a list and a reference to the next cell and a special marker for the end of the list. The basic operations to construct lists are `[]` (pronounced "nil") to create an empty list and `(:)` (pronounced "cons") to append an element to an already existing list. That is, `elt:lst` is the list resulting from putting the value `elt` in front of the list `lst`. So, list literals can also be written as follows:

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
Prelude> 'a' : 'b' : 'c' : []
"abc"
```

The functions that get information about the shape and the contents of the list are `null` to check whether a list is empty: `head`, to get the first element, and `tail`, to get the list without that first element, also known as the rest of the list. Here are some examples of applying these functions:

```
Prelude> null [1,2,3]
False
Prelude> null []
True
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> head []
*** Exception: Prelude.head: empty list
```

Figure 2-1 shows a graphical representation of the operators and functions on lists I have talked about. The `(:)` operator is used to bind together an element with the rest of the list, and you can split those elements apart again using `head` and `tail`. You can also see how a list is a series of cons operations that always end with the empty list constructor, `[]`.

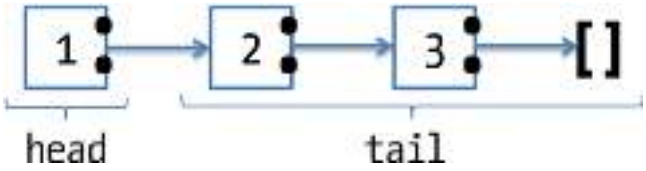

Figure 2-1. *Graphical representation of list constructors and destructors*

If you try to get the head or the tail of an empty list, you get an error, as you may expect. Be aware that exceptions are not the preferred way to handle errors in Haskell (you will see why in more detail in subsequent chapters) and by default make the entire program crash when found. To prevent errors from operations on empty lists, just be sure to check for nonemptiness before applying functions such as `head` and `tail` (or use pattern matching, another syntax that will be introduced shortly).

In fact, looking at the output of `null`, you may have noticed two new values I talked about before: `True` and `False`. These are the only two elements of the `Bool` type, which represent Boolean values. Several standard functions for combining these two values (and &&, or || and `not`) are provided in the Prelude. Most programming languages originating from C, such as C++ and Java, inherit from the former two kinds of Boolean operators. You'll find long-circuiting (& and |) operators, which always evaluate both sides of the expression, and short-circuiting (&& and ||) operators, which may stop after evaluating only one side. In Haskell, because of its lazy evaluation model, these operators always perform their job in the short-circuiting manner. Apart from that, there exists `and` and `or` functions that take a list of Booleans and perform the operations.

```
Prelude> (True && False) || (False && not False)
False
Prelude> or [True, False, and [False, True, True]]
True
Prelude> (2 == 2.1) || (2 < 2.1) || (2 > 2.1)
True
```

■ **Caution** The usual warnings about comparing floating-point values apply here. Computers are not able to represent with exact precision all the values, so you may find that equalities that you expect not to hold actually do. For example, in my system the expression `(4.00000000000000003 - 4) == 0` evaluates to `True`.

Along with these functions, another important construction related to Booleans is `if-then-else`. An expression with the form **`if`** `b` **`then`** `t` **`else`** `f` evaluates to `t` if the value of `b` is `True`, and it evaluates to `f` otherwise. This structure looks similar to the one found in imperative languages but has these important differences:

- `t` and `f` must be expression themselves, in accordance to the rest of the language, because the entire `if` expression must have a value at the end. Writing something like `if True then 1 else 2` is the common pattern in Haskell.
- Both `then` and `else` branches must be present along with the `if`. If this were not the case, then the expression wouldn't be evaluable for some of the values of `b`. Other languages opt to return a default value for the nonexistent `else`, but Haskell makes no commitment.
- The entire expression must have a defined type. The way Haskell manages to ensure that is by forcing both `t` and `f` expressions to have the same type. Thus, an expression such as `if True then 1 else "hello"` won't be accepted by either the compiler or the interpreter.

To make real use of `if` expressions, you need functions that return type `Bool`. This includes the comparison functions between numbers: `==` (equality), `/=` (inequality, but be aware that this function has a different name than in C and Java, where it's called `!=`), `>=` (greater than or equal to), `>` (greater than), `<=` (less than or equal to), and `<` (less than). The following is an example of an `if` expression:

```
Prelude> if 3 < 4.5 then "3 is less than 4.5" else "3 is not less than 4.5"
"3 is less than 4.5"
```

Let's make the interpreter return the head of a list of strings if it is not empty or return `"empty"` otherwise.

```
Prelude> if not (null ["hello","hola"]) then (head ["hello","hola"]) else "empty"
"hello"
Prelude> if not (null []) then (head []) else "empty"
"empty"
```

Lists can contain other lists as elements (or to any level of nesting). As `[τ]` are lists of type `τ`, lists of lists would be `[[τ]]`. The inner lists inside the outer lists need not be of the same length (so they are not equivalent to arrays of multiple dimensions). One important thing to remember is that an empty list can be a member of a larger list of lists, so `[]` and `[[]]` are not equivalent. The first is a completely empty list of lists, whereas the second is a list that contains only one element, which is an empty list.

```
Prelude> :t [['a','b','c'],['d','e']]
[['a','b','c'],['d','e']] :: [[Char]]
Prelude> head [['a','b','c'],['d','e']]
"abc"
Prelude> head (head [['a','b','c'],['d','e']])
'a'
Prelude> head [[]]
[]
```

For sure you have become bored while typing more than once the same constant list in the interpreter. To overcome this, you will learn about the essential ways to reuse functionality across all programming languages: defining functions that work on different input values and creating temporal bindings. But before that, Exercise 2-1 includes some tasks to see whether you have understood the concepts up to this point.

EXERCISE 2-1. LISTS OF LISTS

I have covered a lot of material about the most basic types and expressions in Haskell. The following tasks exercise the knowledge you have gained so far. In all cases, the solutions are expressions that can be typed in the interpreter to check whether they work.

- Rewrite the previous list literals using only `(:)` and the empty list constructor, `[]`.
- Write an expression that checks whether a list is empty, `[]`, or its first element is empty, like `[[],['a','b']]`.
- Write an expression that checks whether a list has only one element. It should return `True` for `['a']` and `False` for `[]` or `['a','b']`.
- Write an expression that concatenates two lists given inside another list. For example, it should return `"abcde"` for `["abc","de"]`.

sample content of Beginning Haskell: A Project-Based Approach

- read online Insurgent (Divergent, Book 2) online
- download Die Zeitpolizei (Perry Rhodan SilberbÃ¤nde, Band 36; M 87, Band 4)
- *Legacies of Plague in Literature, Theory and Film here*
- **download Emerald Star (Hetty Feather)**
- click Contemporary Chinese Vegetarian pdf

- http://xn--d1aboelcb1f.xn--p1ai/lib/The-Gone-Away-World.pdf
- http://bestarthritiscare.com/library/Pure-Dessert--True-Flavors--Inspiring-Ingredients--and-Simple-Recipes.pdf
- http://cavalldecartro.highlandagency.es/library/Coherence--The-Secret-Science-of-Brilliant-Leadership.pdf
- http://aseasonedman.com/ebooks/Emerald-Star--Hetty-Feather-.pdf
- http://aseasonedman.com/ebooks/Contemporary-Chinese-Vegetarian.pdf