About the Author
Kenneth A. Lambert is a professor of Computer Science at Washington and Lee University. He has taught college courses in computer programming for over 30 years.

He is the author, with Douglas Nance and Thomas Naps, of a series of programming textbooks in C++. He is the author, with Martin Osborne, of a programming textbook in Smalltalk and a series of programming textbooks in Java. He is the author, with Kenneth Louden, of a textbook on programming languages. And he is the sole author of a series of programming textbooks in Python.

For more information on Professor Lambert’s publications, teaching, and research interests, visit his Web site at http://home.wlu.edu/~lambertk. For information on this book, including example programs, visit the book’s Web site at http://home.wlu.edu/~lambertk/haskell.
Acknowledgement and Dedication

The inspiration for this book comes from my two wonderful students, Lydia Barit and Maria José Herrera Quesada, who joined me in my initial exploration of functional programming in Haskell. With admiration and gratitude, I dedicate this book to them. I am also grateful to my good friend, Martin Osborne, for providing helpful comments and wise suggestions.
Disclaimer of Warranty
The author makes no expressed or implied warranty of any kind regarding the materials presented in this book or its program examples. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.
Contents

About the Author .................................................................................................................. 3
Acknowledgement and Dedication ....................................................................................... 4
Disclaimer of Warranty .......................................................................................................... 5
Introduction .......................................................................................................................... 9
  Functional Programming ...................................................................................................... 9
  The Haskell Programming Language .................................................................................. 11
  A Note on the History of Functional Programming and Functional Languages .......... 12
Part I. Basic Computations: Data Types, Expressions, and Function Applications 14
  Getting Started with Haskell ............................................................................................ 14
    Exercises ......................................................................................................................... 17
  Basic Data Types: Numbers, Characters, and Booleans ..................................................... 18
    Integers ........................................................................................................................... 18
    Floating-Point Numbers ................................................................................................. 19
    Characters ....................................................................................................................... 19
    Boolean Values ............................................................................................................... 20
  Arithmetic Expressions ...................................................................................................... 20
  Comparison Expressions ................................................................................................... 22
  Logical Expressions ........................................................................................................... 22
  Variables and let Expressions ........................................................................................... 23
    Exercises ......................................................................................................................... 25
  Static Type Checking, Type Inference, and Type Signatures ........................................... 25
  Function Applications ...................................................................................................... 28
  Modules ............................................................................................................................. 30
  Defining Simple Functions ............................................................................................... 32
    Exercises ......................................................................................................................... 34
  Recursive Functions and Pattern Matching .................................................................... 34
    Exercises ......................................................................................................................... 37
  Functions with Guards ..................................................................................................... 38
    Exercises ......................................................................................................................... 39
  if/then/else and case Expressions .................................................................................... 41
    Exercises ......................................................................................................................... 43
  Tail Recursion .................................................................................................................. 44
    Exercises ......................................................................................................................... 47
  where Clauses .................................................................................................................. 48
    Exercises ......................................................................................................................... 49
  Dealing with Errors ........................................................................................................... 49
    Syntax Errors .................................................................................................................. 49
    Semantic Errors ............................................................................................................. 49
    Logic Errors ..................................................................................................................... 50
    Using the error Function ............................................................................................... 50
Part II. Basic Data Structures: Lists, Strings, and Tuples ................................................. 52
  Lists ................................................................................................................................. 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic List Operations</td>
<td>52</td>
</tr>
<tr>
<td>Recursive List Operations</td>
<td>54</td>
</tr>
<tr>
<td>Constructing Lists and Pattern Matching with the : Operator</td>
<td>55</td>
</tr>
<tr>
<td>Appending Lists with ++</td>
<td>57</td>
</tr>
<tr>
<td>Sorting a List</td>
<td>57</td>
</tr>
<tr>
<td>Exercises</td>
<td>58</td>
</tr>
<tr>
<td>Strings</td>
<td>59</td>
</tr>
<tr>
<td>String Comparisons</td>
<td>59</td>
</tr>
<tr>
<td>Using the show Function</td>
<td>60</td>
</tr>
<tr>
<td>Representing Numbers as Bit Strings</td>
<td>61</td>
</tr>
<tr>
<td>Exercises</td>
<td>63</td>
</tr>
<tr>
<td>Tuples</td>
<td>63</td>
</tr>
<tr>
<td>Exercises</td>
<td>64</td>
</tr>
<tr>
<td>List Comprehensions</td>
<td>65</td>
</tr>
<tr>
<td>Exercises</td>
<td>67</td>
</tr>
<tr>
<td>Part III. Functions as Data: Higher-Order Functions and Function-Level Operators</td>
<td>68</td>
</tr>
<tr>
<td>Mapping</td>
<td>68</td>
</tr>
<tr>
<td>Lambda Expressions</td>
<td>69</td>
</tr>
<tr>
<td>Exercises</td>
<td>70</td>
</tr>
<tr>
<td>Filtering</td>
<td>70</td>
</tr>
<tr>
<td>Exercises</td>
<td>71</td>
</tr>
<tr>
<td>Folding</td>
<td>71</td>
</tr>
<tr>
<td>Folding from the Right</td>
<td>72</td>
</tr>
<tr>
<td>Folding from the Left</td>
<td>73</td>
</tr>
<tr>
<td>Exercises</td>
<td>74</td>
</tr>
<tr>
<td>Lazy Evaluation</td>
<td>74</td>
</tr>
<tr>
<td>Lazy Days</td>
<td>74</td>
</tr>
<tr>
<td>Infinite Lists</td>
<td>74</td>
</tr>
<tr>
<td>Partial Function Application</td>
<td>76</td>
</tr>
<tr>
<td>Currying</td>
<td>77</td>
</tr>
<tr>
<td>Function Level Operators: Application and Composition</td>
<td>77</td>
</tr>
<tr>
<td>Part IV. Creating New Data Types</td>
<td>79</td>
</tr>
<tr>
<td>Algebraic Types</td>
<td>79</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>79</td>
</tr>
<tr>
<td>Product Types</td>
<td>82</td>
</tr>
<tr>
<td>Union Types</td>
<td>84</td>
</tr>
<tr>
<td>Exercises</td>
<td>86</td>
</tr>
<tr>
<td>Record Types</td>
<td>87</td>
</tr>
<tr>
<td>Exercises</td>
<td>88</td>
</tr>
<tr>
<td>Part V. Generic Data Types and Type Classes</td>
<td>89</td>
</tr>
<tr>
<td>Generic Types</td>
<td>89</td>
</tr>
<tr>
<td>Exercises</td>
<td>90</td>
</tr>
<tr>
<td>Type Classes</td>
<td>90</td>
</tr>
<tr>
<td>Exercises</td>
<td>94</td>
</tr>
<tr>
<td>Part VI. Interacting with the World: Terminal I/O and Files</td>
<td>95</td>
</tr>
<tr>
<td>Interactive Input and Output</td>
<td>95</td>
</tr>
</tbody>
</table>
Introduction
Everyone should learn how to code.¹ Computer programming has joined reading, writing, and arithmetic as one of the basic skills of an educated person in the 21st century. Millions of people now use computers to get things done, whether on their jobs or in their lives away from work. Computer programming takes your use of computers a step further, not only by adding to the useful things a computer can do for you, but by sharpening your mental skills to solve problems that do not require computers.

This book introduces programming in a programming language called Haskell. Haskell was designed to support a style of programming called functional programming. The meaning of this term will soon become clear, but first a word of warning. This book is intended primarily for readers who already know a bit about programming, through experience with programming languages such as Java, Python, or C++. These readers will benefit by learning the functional style of programming as a new way of solving problems with a computer. If you know nothing about programming, and wish to have an easier go of it, you might consider trying your hand at programming in a language like Java or Python before you go through this book. Some excellent introductory texts in these languages are listed in the bibliography.

Functional Programming
A program is a set of instructions that transforms a set of input values into a set of output values. These values or data can be of any type, such as numbers, text, images, or sound clips. Here are three common examples of programs:

- A pocket calculator allows you to compute the sum or product of two numbers.
- A text analysis program allows you to obtain a word count of a document.
- An image manipulation program allows you to convert a color image to a grey scale image.

When you look inside a functional program, all that you see are other, smaller functional programs that collaborate in some way to solve a problem. These subprograms are called functions. Functions are either provided directly by the programming language or are defined by the programmer.

You can think of a function as a box into which you feed one or more input values and receive an output value. For example, consider the computation of the square of the number 3, which produces 9, written as the equation $3^2 = 9$ in mathematics. Let’s assume that you have a function named `square`, whose definition is

```
square x = x * x
```

This definition says that the square of any number $x$ is $x$ multiplied by $x$. You can compute $3^2$ by calling the function with the expression `square 3`, as shown in Figure 1.

---

¹ As stated by Suzanne Keen, the Thomas Broadus Professor of English and Dean of the College at Washington and Lee University.
When you evaluate the function \textit{square} 3 and receive a result of 9, here is what happens, as illustrated in the diagram:

1. The input value, the number 3, is passed into the \textit{square} function’s box.
2. Within this box, the number 3 is passed, as each of two inputs, into the box for the multiplication function, labeled \( \ast \).
3. This box, which you cannot open, produces the value 9, which is then passed along as the value produced by the \textit{square} function’s box.

This example might seem too trivial to discuss. But the important point is that any function behaves in a similar manner. When you engage in functional programming, you view the behavior of a program as the work of functions \textit{all the way down}. This means that, beginning with the top-level program, whenever you open a function box to look inside, you will see other collaborating functions, until you reach the boxes for the built-in functions of the language, which you cannot open. In functional programming, the programmer need only think of the type of input data (numbers, text, images, sound clips, or whatever), the type of output data, and the arrangement of functions that will transform the given input data into the desired output data.

Of course you don’t have to open a function box to see how it works if you just want to use it to perform a computation. Consider the problem of computing the hypotenuse of a right triangle, given the other two sides \( a \) and \( b \), according to the mathematical equation

\[ hypotenuse = \sqrt{a^2 + b^2} \]

Assuming that you have at your disposal the \textit{square} function defined earlier, as well as the built-in functions \texttt{sqrt} for square root and \texttt{+} for addition, you can define a new function named \texttt{hypotenuse}, as follows:

\[ \text{hypotenuse} \ a \ b = \text{sqrt} \ (\text{square} \ a \ + \ \text{square} \ b) \]

This definition says that the hypotenuse of any right triangle with the sides \( a \) and \( b \) is the square root of the sum of the squares of \( a \) and \( b \). The evaluation of this function with the sides 3 and 4, \texttt{hypotenuse 3 4}, is diagrammed in Figure 2.
FIGURE 2
Computing the hypotenuse of a right triangle with sides 3 and 4

Note that this diagram opens the box for the hypotenuse function to show how it works, but keeps the boxes for the square function closed. At this point, you need not know anything more about the square function than you know about the sqrt, + and * functions, except that it will produce the expected outputs for any of its possible inputs.

You will soon learn that functional programming has many benefits as well as some costs. For now, there are three important points to remember about functional programming:

1. Programs consist of cooperating functions, as do functions.
2. A function transforms input data into output data.
3. A function always produces the same outputs when given the same inputs.

The Haskell Programming Language
The Haskell programming language is named for the mathematician and logician Haskell Curry (1900-1982). The principal ideas and the design of the language were the work of a committee that met at a conference on functional programming in 1987, and the first definition of the language was released in 1990. The committee believed that the most useful aspects of various functional languages should be consolidated into a single standard, and Haskell was the result.

Haskell supports the aspects of functional programming discussed earlier. In addition to support for functions that transform data into other data, the language includes features that you will explore in depth in this book but will just be mentioned for now:

- **Strict, static type checking.** This means that the compiler checks the types of all data and disallows any attempts to perform computations on data of an inappropriate type (like passing a string as data to the square function mentioned earlier). Thus, type incompatibility errors are caught at compile time, rather than run time, as they are in other languages like Python.
- **Higher-order functions and functions as first-class data.** Higher-order functions are functions that take other functions as arguments and possibly return functions as values. They are useful for coordinating the work of cooperating functions. Functions themselves are data values. The compiler checks the types of functions just like those of any other data.
- **Function-level operators for composing functions.** These operators allow the programmer to cleanly glue together functions within a common data path.

- **Pattern matching.** This provides a means of selecting different computations depending on the structure of the data in a function’s arguments.

- **Polymorphic functions.** These are functions that provide a single operation on data of different types, such as lists of integers and lists of strings.

- **Type classes.** A type class provides a set of functions that can be defined differently for different data types belonging to that type class. For example, the function that determines whether one value is less than another must use a different method for two integers than it does for two strings.

The nature and use of all of these features will become clear as you proceed through this book. But before turning to that, this introduction concludes with a brief history of functional languages and functional programming.

**A Note on the History of Functional Programming and Functional Languages**

A type of language to describe functions was first thoroughly investigated by the mathematician and logician Alonzo Church (1903-1995). In 1936, several years before the invention of the first electronic digital computers, Church published a paper in which he developed a symbolic language for expressing computation in terms of functions.² The purpose of this language was to assist Church in exploring and proving properties of formal symbol systems in mathematical logic. The language he developed for that, called the *recursive lambda calculus*, was to become the basis several decades later for the development of the first functional programming languages.

The first programming language based in large part on Church’s functional language theory is called LISP, which stands for LIst Processing Language. Created by the computer scientist John McCarthy (1927-2011) at MIT in the late 1950s, LISP is the second oldest programming language still in widespread use. The oldest such language is called FORTRAN, short for FORmula TRANslation language. Unlike FORTRAN, which was intended for numeric applications in science and engineering, LISP was intended for writing functions that transform symbolic information, represented as lists of symbols, into other symbolic information. McCarthy included many of the language elements from Church’s theory, such as recursive functions, conditional expressions, and functions that coordinate the computations of other functions (you will learn more about these features in the chapters that follow). McCarthy also devised a program, called an *interpreter*, to translate functional programs written in LISP to a form that a real computer can execute.

Although LISP was regularly used for symbolic programming in such areas as artificial intelligence, there are two reasons why LISP and the functional model of programming did not immediately catch on in the wider programming community:

---
1. The LISP interpreter required computer hardware resources, such as substantial memory and a fast processor, that were beyond the budgets of most programmers.

2. Because most programmers were trained in FORTRAN or its descendants, they did not see the benefits of the functional model of computation.

Over the next three decades, three developments occurred that allowed programmers to understand and begin to enjoy the benefits of functional programming:

1. The benefits of functional programming were clearly compared to the costs of other styles of programming.
2. The cost of memory and processor speed declined, and fast, inexpensive compilers and runtime systems for functional programming became available to programmers.
3. New functional languages, such as Haskell, which represented some improvements on the earlier approaches based on LISP, began to come into play.

The challenge of articulating the benefits of functional programming was taken up in the late 1970s. John Backus (1924-2007), the computer scientist who had invented FORTRAN decades earlier and who had developed one of the first grammatical notations for describing programming languages, won the Association for Computing Machinery’s annual Turing Award (the equivalent of a Nobel prize in computer science) in 1977. In his Turing Award Lecture, which he published in 1978, Backus argued that the conventional model of computation was rapidly becoming unwieldy for the development of large-scale software systems, and that a new approach was needed.

Backus argued that the functional model provides not only a simpler way to organize programs but also a way to reason mathematically about their correctness. He emphasized the safety and consistency of the functional model, in which the basic functional elements simply transform data values into other data values. He also argued that the means of organizing and combining these elements should adhere to the functional model.

The developers of functional programming languages in the 1980s and 1990s took these arguments seriously, and created languages that reflect the functional model in a very pure manner. When these languages became available and the programming community began to see their advantages, functional programming became an important area of study in university curricula. Functional programming has also recently come into use in such mission-critical areas as the financial sector and the scientific community.

For more background information on functional programming and the Haskell programming language, consult the bibliography of this book or the Haskell website at https://www.haskell.org/. Now it’s time to learn functional programming in Haskell!

---

Part I. Basic Computations: Data Types, Expressions, and Function Applications

The first part of this book introduces you to the basic elements of Haskell programs, as well as the programming environment in which you can experiment with them.

Getting Started with Haskell

Programming environments typically consist of a text editor for writing the programs, a compiler for translating them to executable form, and a run time system for running or executing them.

The text editors that come with most computer systems are adequate for beginners, but a free, third-party editor, such as Editra (http://www.editra.org/), with features for formatting and numbering lines of code is recommended.

The other two software components you will need are the Glasgow Haskell Compiler (the GHC) and the Glasgow Haskell Compiler Interactive (the GHCi). You can obtain these components in one package, called the Haskell Platform, from the Haskell website at https://www.haskell.org/downloads. You should be careful to download the Haskell Platform for your particular operating system (Linux, OS X, or Windows).

Installation of the Haskell Platform is straightforward on all three systems, but Mac users should first upgrade their OS and install Apple’s command line tools. To do this, you launch the Terminal app, found in /Applications/Utilities/, run the command

```
xcode-select -install
```

and follow the instructions.

Documentation for the Haskell programming language and its standard libraries can be found at https://www.haskell.org/documentation.

Haskell code is developed in two ways:

1. Interactively within the GHCi. This is an interpreter that allows you to enter simple, one-line Haskell expressions and view the results.
2. Edit a file of Haskell code in a text editor and then compile and run the code. These files are called scripts or modules. You can compile and run a script or module by loading it into the GHCi. Alternatively, you can compile the script or module from the terminal. Then, for certain special kinds of scripts, you can run the code as a standalone program in a terminal.

This book uses both methods: the GHCi is useful for trying out short code segments, and an editor is useful for developing modules and complex programs.

To launch the GHCi, you open a terminal on Mac or Linux systems and enter the ghci command, or select the appropriate Start menu command if you are a Windows user. Figure 3 shows a Mac terminal window running the GHCi, where several Haskell expressions have been entered.
In the GHCi, you enter each expression at a Prelude> prompt and view the results. More precisely, whenever you hit the return key, the GHCi reads the expression, evaluates it, and prints the result. If all goes well, a new prompt appears below the result; otherwise, if an error occurs, the GHCi displays an error message. This process is also called the REPL (short for read-evaluate-print-loop).

The first two expressions in Figure 3 perform arithmetic and return numbers, while the last two evaluate a string and print some text. Note that the :quit command exits the interpreter and returns the programmer to the terminal window. There are several special interpreter commands, all beginning with the : character, which you can review by entering the :help command. Table 1 provides a short list of commonly used GHCi commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>:browse[!] [[*]&lt;mod&gt;]</td>
<td>Display the names defined by module &lt;mod&gt; (&lt;!: more details; *: all top-level names)</td>
</tr>
<tr>
<td>:cd &lt;dir&gt;</td>
<td>Change directory to &lt;dir&gt;</td>
</tr>
<tr>
<td>:edit &lt;file name&gt;</td>
<td>Edit &lt;file name&gt;</td>
</tr>
<tr>
<td>:help, :?</td>
<td>Display all commands</td>
</tr>
<tr>
<td>:load &lt;module&gt;</td>
<td>Load &lt;module&gt; and its dependents</td>
</tr>
<tr>
<td>:quit</td>
<td>Quit the GHCi</td>
</tr>
<tr>
<td>:type &lt;expression&gt;</td>
<td>Display the type of &lt;expression&gt;</td>
</tr>
<tr>
<td>:!&lt;command&gt;</td>
<td>Run a shell command</td>
</tr>
</tbody>
</table>

In addition to these commands, there are several shortcuts. For example, the arrow keys can be used to move back and forth through the history of expressions that have already
been entered. This is a handy way to avoid retyping if you want to make just minor changes to a piece of code and run it again. Also, the expression named it displays the value most recently computed in the GHCi.

The second way to develop and run Haskell code is to

1. Use a text editor to create a file of code, called a module or a script
2. Compile this file of code using the GHC (Glasgow Haskell Compiler)
3. Run the code, either by loading it into the GHCi or, in certain special cases, by launching it from a terminal window

Although this method is used when the code becomes longer and more complex than a single, one-line expression, a brief example will illustrate the technique.

In Figure 3 you saw the expression putStrLn "Hello world!" entered in the GHCi. You can turn this code into a script or standalone program by entering the following text in a text editor window and saving the file as HelloWorld.hs (Haskell source file names are spelled in CamelCase and end with a .hs extension). Figure 4 shows the code for this short Haskell script in an Editra window.

FIGURE 4
Editing a Haskell script in a text editor

This code defines a module named Main that includes a function named main. When the program is launched in a terminal window, the runtime system looks for a module named Main and begins execution by running the module’s main function. This function, in turn, prints the string “Hello world!” in the terminal window.

Note the following points about this code:
- Lines 1-12 of code in grey are *program comments*. This text is not program code, but exists to enlighten the reader about the purpose of the program.
- The terms highlighted in brown in line 14 are *reserved words*, which serve special purposes in the code.
- The *string* in line 17 is highlighted in magenta.
- Haskell code is *case sensitive*. Therefore, the name `Main` can mean something different than the name `main`.

To compile and run this script in a terminal window, you run the following terminal commands:

```bash
$ ghc --make HelloWorld
[1 of 1] Compiling Main             ( HelloWorld.hs, HelloWorld.o )
Linking HelloWorld ...
$ ./HelloWorld
Hello world!
$ 
```

Alternatively, you can load and run this program in the GHCi, as shown in the following session:

```haskell
Prelude> :load HelloWorld
[1 of 1] Compiling Main             ( HelloWorld.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Hello world!
*Main>
```

It’s not important that you understand the details of the program code and the way that the programming environment works at this point; all this will become clear as you work through this book. But now you know enough to develop some simple Haskell programs.

**Exercises**

Launch the GHCi, enter the following Haskell expressions, one line at a time, and view the results. Try to guess what is going on from the results. If you make a mistake and see an error message, just try to enter the expression again. Don’t skim these items with your eyes; enter them with your hands in the GHCi!

```haskell
34
:type 34
3 + 4 * 5
(3 + 4) * 5
it
14 / 4
14 `div` 4 -- The back quote is below the ~ character on the keyboard
14 `mod` 2
2 == 2
2 == 3
2 < 3
2 > 3
2 <= 3
2 /= 3
:type 6.7
```
round 6.7
sqrt 2
pi
not True
:type True
not (2 == 3)
True && False -- && means and
True && True
True || False -- || means or
False || False
'a'
:type 'a'
'a' < 'b'
fromEnum 'a'
toEnum 98 :: Char
'a' == 'A'
'a' < 'A'
fromEnum 'A'

Basic Data Types: Numbers, Characters, and Booleans
If you have not done the first set of exercises, go back and do so now! In the next few sections, you will work bottom up to learn the basic elements of Haskell expressions. The first such elements are the basic data values and their types. The data values shown in the last set of exercises belong to these types. They are listed in Table 2. Note that the names of Haskell data types are capitalized.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Example Values</th>
<th>What It Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>True, False</td>
<td>The two truth values.</td>
</tr>
<tr>
<td>Char</td>
<td>'a', '!'</td>
<td>The set of characters (on the Latin keyboard and others as well).</td>
</tr>
<tr>
<td>Double</td>
<td>3.14, 0.001</td>
<td>The set of double-precision floating-point numbers.</td>
</tr>
<tr>
<td>Float</td>
<td>3.14, 0.001</td>
<td>The set of single-precision floating-point numbers.</td>
</tr>
<tr>
<td>Int</td>
<td>0, 67, 1000000</td>
<td>The set of integers ranging from $-2^{31}$ to $2^{31} - 1$.</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td>Includes the Ints as well as greater magnitudes A range of integers limited only by computer memory.</td>
</tr>
</tbody>
</table>

Integers
The Int and Integer types represent whole numbers. The set of the whole numbers is not infinite, as in mathematics, but is limited by the memory available to represent large magnitudes on a real computer. The values of the Int type are a subset of the values of the Integer type. The Int type is used to represent small integers that use less memory than the large integers in the Integer type. The values of the Int type range from $-2^{31}$ to
The range of values of the `Integer` type is bounded only by the memory of your computer.

**Floating-Point Numbers**

The `Float` and `Double` types represent floating-point numbers. Floating-point numbers provide an approximation of the real numbers (numbers with a decimal point) in mathematics. Thus, the set of floating-point numbers is not the same as the set of real numbers. The latter numbers have infinite precision as well as an infinite range of magnitudes. The precision (the number of digits to the right of a decimal point) and the magnitude of a floating-point number are both limited by computer memory. The `Double` type can represent a greater precision and a larger magnitude than the `Float` type, at the cost of twice the memory.

**Characters**

The set of characters forms the basis for representing text. Three traditional character sets are

- The ASCII (American Standard Code for Information Interchange) set, which consists of 128 characters on the Latin keyboard, such as upper and lowercase letters, digits, punctuation marks, whitespace, and some non-printing control characters.
- The extended ASCII set, which adds 128 special characters to the basic ASCII set.
- The Unicode set, which consists of $2^{16}$ (or 65,536) characters, including the extended ASCII set as well as characters and symbols from non-Latin alphabets, such as Arabic, Greek, Cyrillic, and Chinese. Like most contemporary languages, Haskell supports the Unicode set.

The characters are not a numeric type, so you cannot do arithmetic with them. However, you can compare them for lexicographic order (‘a’ is less than ‘b’). Like the integers, the characters are an ordinal type, meaning that the set is ordered from the smallest value to the largest.

As you saw from running the `fromEnum` and `toEnum` functions in the exercises, each character maps to an associated integer or ordinal value underneath the hood. Table 3 shows the original ASCII character set and the associated ordinal values. Note that some common sequences, such as the uppercase letters, the lowercase letters, and the digits, occupy consecutive ordinal positions. But also note that the letter ‘a’ (ASCII 97) has a different ordinal value than the letter ‘A’ (ASCII 65), and that the sequence of uppercase letters is completely separate from the sequence of lowercase letters in the ASCII set.
**Boolean Values**

The `Bool` type represents the Boolean values. The Boolean values are named for George Boole, one of the founders of modern logic. There are only two Boolean values, named `True` and `False` in Haskell. These are the values of conditions that control the flow of execution in a program. For example, the equality operator `==` compares two values and returns `True` if they’re equal or `False` otherwise. You could use this operator to test a number to see if it’s nonzero before dividing by it.

**Arithmetic Expressions**

The GHCi can behave like a simple calculator, in which you enter arithmetic expressions and view the results. Haskell supports simple infix notation for these expressions, where the operator occurs between its two operands, as in `2 + 3`. The spaces surrounding the operator are helpful to the reader but not necessary.

In the absence of parentheses, standard operator precedence applies, so exponentiation is done before multiplication, which is done before addition. Thus, the expression `2 + 3 * 4 ^ 5` evaluates to the same result as `(2 + (3 * (4 ^ 5)))`, rather than `(((2 + 3) * 4) ^ 5)`. In the second expression, the parentheses override the standard precedence. Table 4 lists the binary arithmetic operators in increasing order of precedence.

**TABLE 3**

The Original ASCII Character Set

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
</tr>
<tr>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
</tr>
<tr>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
</tr>
</tbody>
</table>
| RS  | US  | SP  | !   | "   | #   | $   | %   | &   | `
| (   | )   | *   | +   | ,   | -   | .   | /   | 0   | 1 |
| 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | :   | : |
| <   | =   | >   | ?   | @   | A   | B   | C   | D   | E |
| F   | G   | H   | I   | J   | K   | L   | M   | N   | O |
| P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y |
| Z   | [   | \   | ]   | ^   | _   | `   | a   | b   | c |
| d   | e   | f   | g   | h   | i   | j   | k   | l   | m |
| n   | o   | p   | q   | r   | s   | t   | u   | v   | w |
| x   | y   | z   | {   | |   } | ~   | DEL |   |   |

**TABLE 4**

Some Haskell Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>What It Is</th>
<th>Example Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>Addition and subtraction.</td>
<td><code>5 + 2</code></td>
<td>7</td>
</tr>
<tr>
<td>* , /</td>
<td>Multiplication and division.</td>
<td><code>5 / 2</code></td>
<td>2.5</td>
</tr>
<tr>
<td>div, mod</td>
<td>Integer quotient and remainder.</td>
<td><code>5 </code>div<code> 2</code></td>
<td>2</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation.</td>
<td><code>5 ^ 2</code></td>
<td>25</td>
</tr>
</tbody>
</table>
Note that some operators, such as integer quotient (\texttt{div}) and remainder (\texttt{mod}) use names rather than operator symbols. In these cases, the name must be enclosed in back quotes when used in an infix expression.

To negate a number, you can use the unary negation operator \texttt{-} or the \texttt{negate} function. Thus, the following two expressions produce the same result:

\begin{verbatim}
Prelude> -34
-34
Prelude> negate 34
-34
\end{verbatim}

You must be careful to enclose a unary negation in parentheses when it’s used in the context of a binary arithmetic expression, as in

\begin{verbatim}
Prelude> 5 * (-2)
-10
\end{verbatim}

Another thing to be aware of when doing arithmetic is that the type of the result value depends on the types of the operands. With the exception of the / operator, which always produces a floating-point result, the use of two integers produces an integer result, and the use of at least one floating-point number normally produces a floating-point result.

Of course, any attempt to perform arithmetic with an operand that is not a number produces an error, as shown in this attempt to add a Boolean value to an integer:

\begin{verbatim}
Prelude> 4 + True
<interactive>:3:3:
   No instance for (Num Bool) arising from a use of ‘+’
   In the expression: 4 + True
   In an equation for ‘it’: it = 4 + True
\end{verbatim}

Occasionally you might receive an error message when mixing together integers and floating-point numbers in arithmetic expressions. In those cases, you can call the \texttt{fromIntegral} function with the integer as an argument to remedy the problem:

\begin{verbatim}
Prelude> 0.5 * fromIntegral 2
1.0
\end{verbatim}

If you want a result that’s rounded to the nearest whole number, you can use the \texttt{round} function, as follows:

\begin{verbatim}
Prelude> 5 `div` 2
2
Prelude> 5 / 2
2.5
Prelude> round 5 / 2
<interactive>:2:1:
   No instance for (Fractional a0) arising from a use of 'it'
   The type variable 'a0' is ambiguous
\end{verbatim}
Note: there are several potential instances:

```haskell
instance Integral a => Fractional (GHC.Real.Ratio a)
  -- Defined in 'GHC.Real'
instance Fractional Double -- Defined in 'GHC.Float'
instance Fractional Float -- Defined in 'GHC.Float'
```

In the first argument of `print`, namely 'it'

In a stmt of an interactive GHci command: print it

Prelude> round (5 / 2)
3

Note the use of parentheses in the last expression. Because function applications have a higher precedence than binary operators, you must use parentheses to guarantee that the `round` function is applied to the entire expression `5 / 2`. The error shown on the previous entry, `round 5 / 2`, occurs because the GHci thinks you mean to supply three arguments to the `round` function, when only one argument is expected.

**Comparison Expressions**

A *comparison expression* tests two values of the same type for the presence of a relationship (such as equality), and returns `True` or `False`, indicating whether those values have that relationship or not. A wide range but not all types of values can be compared for equality or inequality. Only ordinal types of values can be compared for less than or greater than relationships as well. Table 5 lists the comparison operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>What It Is</th>
<th>Example Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equality</td>
<td>5 == 2</td>
<td>False</td>
</tr>
<tr>
<td><code>/=</code></td>
<td>Inequality</td>
<td>5 /= 2</td>
<td>True</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>5 &lt; 2</td>
<td>False</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>5 &gt; 2</td>
<td>True</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
<td>5 &lt;= 5</td>
<td>True</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
<td>5 &gt;= 2</td>
<td>True</td>
</tr>
</tbody>
</table>

Because integers, floating-point numbers, characters, and Boolean values belong to ordinal types, all of the comparison operators can be used with them. However, while you can compare any type of number to any other type of number, you can only compare characters to characters and Boolean values to Boolean values. Later in this book, you will learn that character-level comparisons form the basis for comparing words and lines of text.

**Logical Expressions**

The final category of operations on a basic data type includes the *logical expressions*. The logical operators, traditionally called AND, OR, and NOT, expect Boolean values as their operands (NOT, like arithmetic negation, expects a single operand). Logical expressions return Boolean values. The Haskell logical operators for AND, OR, and NOT are `&`, `||`, and `not`, respectively.
The rules for evaluating logical expressions can be completely specified in a *truth table*, as shown in Table 6.

**TABLE 6**
A Truth Table for the Logical Expressions

|   |   | **not a** | **a && b** | **a || b** |
|---|---|-----------|------------|------------|
| False | False | True | False | False |
| False | True | True | False | True |
| True | False | False | False | True |
| True | True | False | True | True |

Note that the first two columns hold each possible combination of the values of the operands *a* and *b*. You should try out some of these combinations in GHCi to verify the rules.

Most code does not include logical expressions just like these. Instead, a program might evaluate one or two comparisons, and apply a logical operator to the Boolean results. For example, suppose you want to determine whether an integer value *x* lies within a given range of integers. The lower and upper bounds of this range are named *lower* and *upper* (you will see how to give names to values in the next section). Then, the logical expression

\[
\text{lower} \leq x \text{ && } x < \text{upper}
\]

returns *True* if *x* lies within this range specified by *lower* and *upper*, or *False* otherwise. Note that the precedence of the logical operators is less than the precedence of the comparison operators, so parentheses are not required.

This concludes the survey of Haskell’s basic data types and the operations on them. The next two sections show how to give names to expressions and how to perform other operations, called *function applications*.

**Variables and let Expressions**

As you develop more complex and interesting programs, you will find it convenient to give names to values, so you just have to remember the name and not the particular value. For example, the expression \(0.5 \ast 10 \ast 4\) computes the area of a right triangle, but you would not likely guess that just by looking at the numbers. But the expression \(\text{oneHalf} \ast \text{base} \ast \text{height}\) should be immediately recognizable to anyone who has studied basic geometry. In this section, you will learn how to give names to values and use them in Haskell expressions.

As in mathematics, the names given to values are called *variables*. In Haskell, one way to give a name to a value is to use a *let* expression. When used in the GHCi, the form of a *let* expression is

\[
\text{let } \text{<variable name>} = \text{<expression>}
\]
In syntactic forms like this one, some symbols, like `let` and `=`, must literally appear in an expression of that form. The symbols in angle brackets, like `<variable name>`, indicate a type of expression or value that must appear in the expression.

Like other expressions, the `let` expression must occupy just one line in the GCHi. Once you have used a `let`, you can enter the variable to view its value, or use it in subsequent expressions. Here is an example session that uses three variables to compute the area of a right triangle:

```
Prelude> let oneHalf = 0.5
Prelude> let base = 10
Prelude> let height = 4
Prelude> let area = oneHalf * base * height
Prelude> area
20.0
```

Note the following points:

- A `let` expression does not appear to return a value. Its sole purpose is to give a name to a value, so from that point on, that name will refer to that value.
- A `let` expression can be used only once with a given name. Once a name has a value, that name’s value cannot be changed. Thus, although a `let` expression looks like an assignment expression in some other languages, you may think of it as a one-time only binding of a given variable to a value in Haskell.
- Variable names always begin with a lowercase letter. Uppercase letters and digits may be used in a variable name for clarity, as in `interestRate` and `top10Teams`.

Some variables, such as `pi`, have predefined meanings in Haskell:

```
Prelude> pi
3.141592653589793
```

You can see from this example why using a variable would be better than trying to remember a value.

There is one other important restriction on variables. A few words in Haskell, such as `module`, `where`, `if`, `then`, and `let`, are reserved for various purposes. Also, many names already belong to standard variables and functions, like `pi` and `round`. Any attempt to reuse these names for your own variables will result in an error message.

The `let` expression has a more complex and powerful form when used in Haskell scripts with function definitions, as you will see later in this book. For now, if you want to give names to values in a Haskell script, you should do so by omitting the reserved word `let`, as shown in the next script:
{-
File: Triangle.hs

Computes the area of a right triangle
-}

module Triangle where

oneHalf = 0.5
base = 10
height = 4
area = oneHalf * base * height

When you load this module into the GHCi, you can look up the value of area:

Prelude> :load Triangle
Triangle> area
20.0

Exercises
Launch the GHCi, enter the following Haskell expressions, one line at a time, and view the results. Try to guess what is going on from the results. If you make a mistake and see an error message, just try to enter the expression again. Don’t skim these items with you eyes; enter them with your hands in the GHCi!

let radius = 8
let diameter = radius * 2
let circumference = diameter * pi
let area = pi * radius ^ 2
let volume = 4 / 3 * pi * radius ^ 3
(radius, diameter, circumference, area, volume)

Static Type Checking, Type Inference, and Type Signatures
As mentioned in the Introduction, Haskell supports static type checking. When the compiler translates your Haskell source code to machine code, it checks the types of all operands to ensure that they are allowed for the operations to be performed on them. Thus, any potential error, such as an attempt to add a Boolean value to an integer, will be caught at compile time, before the code can be run. As a consequence, your programs will have a greater chance of running correctly, even though some logic or design mistakes might still be present. You will learn more about finding, repairing, and avoiding design errors later in this book.

You have already seen examples of how the GHCi catches type errors and responds with error messages when expressions contain values of the inappropriate type. Now that you know about variables, you might wonder how Haskell enforces static type checking on them as well. Consider the errors in the following GHCi session:

Prelude> let aNumber = 5
Prelude> let aBoolean = True
Prelude> aNumber + 2
7
Prelude> aBoolean + 2
No instance for (Num Bool) arising from a use of '+'
In the expression: aBoolean + 2
In an equation for 'it': it = aBoolean + 2

Prelude> odd aNumber
True
Prelude> odd aBoolean

No instance for (Integral Bool) arising from a use of 'odd'
In the expression: odd aBoolean
In an equation for 'it': it = odd aBoolean

The GHCi obviously knows the types of the variables as well as the types of the values in expressions. A quick check of the types of the variables aNumber and aBoolean verifies that this is the case:

Prelude> :type aNumber
aNumber :: Num a => a
Prelude> :type aBoolean
aBoolean :: Bool

In this example, the GHCi determines the data type of the two variables by a process known as type inference. The GHCi sees that values of given types are given names in the let expressions, so the names pick up those types. However, you might expect to see the type Int instead of the type Num => a -> a, for the variable aNumber, which refers to the value 2. A bit of explanation is in order at this point.

The GHCi :type command returns the type signature of its argument. This argument can be any expression. A type signature includes the expression, followed by a double colon, followed by the data type information, as follows:

<expression> :: <type information>

In the case of the Boolean variable aBoolean, the GHCi has enough information from the value True to assign the variable aBoolean the exact type Bool. But in the case of the variable aNumber, whose value is 2, there is not enough information to assign it such a specific type. This value can belong to any of the many integer types (such as Int, Integer, Int16, and so forth), as well as some other numeric types. So, the variable aNumber receives the type signature, Num a => a, which means any type a that belongs to the class of types called Num. You will learn more about type classes like Num and type variables like a later in this book. Nonetheless, there is still enough information in this type signature to allow the use of the variable aNumber in certain expressions and disallow it in others.

The reason that the variable aBoolean cannot be used with the + or odd operations can be seen by examining their type signatures:

Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude> :type odd
odd :: Integral a => a -> Bool
Note that an operator must be parenthesized when it appears as argument. The type signature of a function or an operator includes the type information of the argument(s) as well as the returned value. The simplest form of this notation is

\[ \text{<function> :: <argument type> -> ... -> <return type>} \]

The type signature of + shows two argument types and a return type (the last three \texttt{a} symbols), all of which must belong to the \texttt{Num} type class. The easiest way to read the type signature \((+) :: \text{Num } a \Rightarrow a -> a -> a\) is to say that the + operator takes a value of type \texttt{a} into a value of type \texttt{a} into a value of type \texttt{a}, where type \texttt{a} belongs to the \texttt{Num} type class. Since \texttt{Bool} does not belong to this type class, an error is flagged for the expression \texttt{aBoolean + 2}.

The signature of the \texttt{odd} function shows one argument whose type must be a member of the \texttt{Integral} type class. As the name implies, this type class includes all of Haskell’s integer types, but not the floating-point types or the \texttt{Bool} type.

Type inference works well when you are experimenting in the GHCi, but when you write scripts in files, you should declare the types of all variables and functions with explicit type signatures in your code. The next example adds type signatures for the variables to the \texttt{Triangle} module discussed earlier:

{-
File: Triangle.hs
Computes the area of a right triangle
-}

module Triangle where

-- The four variables and their types
oneHalf, area :: Float
base, height :: Integer

oneHalf = 0.5
base = 10
height = 4
area = oneHalf * base * height

If you omit the type signatures from this script, the types of the variables are inferred, as they are in the GHCi. But including them

- adds clarity to the code
- helps the Haskell compiler verify that the program is type-safe, because sometimes type inference is impossible
- guarantees that the most specific data type will be used for the problem at hand

Although it’s not necessary in the GHCi, you can include type signatures when you introduce variables there as well. Here is an example:

\texttt{Prelude> let aNumber = 5 :: Integer}
\texttt{Prelude> :type aNumber}
\texttt{Integer}
The GHCi now knows that the type of `aNumber` is `Integer`, because you have specifically declared it to be so in the `let` expression. Note that the type information is not written on a separate line, as in a module file, but follows the definition of the variable in the `let` expression.

Beginners sometimes find the details of static typing arcane, tedious, and annoying, but it’s important to master them early. You will be amply rewarded when you don’t have to track down arcane type errors in complex programs at run time.

**Function Applications**

As discussed in the Introduction, functions and data form the basic building blocks of functional programs. In a functional language like Haskell, functions can be treated like any other data values, or they can be applied to other data – their arguments – to produce new data. In this section, you explore how to express function applications in program code.

Each of the basic arithmetic, comparison, and logical operations examined earlier is actually a function application. The infix notation used with the operators in those expressions is *syntactic sugar* for a prefix form (in prefix, the operator or function appears before its operands). Here are some examples of both types of notations:

```
Prelude> 5 `div` 2  -- infix quotient
    2
Prelude> div 5 2  -- prefix quotient
    2
Prelude> 2 + 5  -- infix addition
    7
Prelude> (+) 2 5  -- prefix addition
    7
```

Note that `div`, which is a function name, must be back quoted when used in an infix expression. The `+` symbol, which is an operator, must be parenthesized when used in a prefix expression.

Most applications of named functions, like `round` and `negate`, appear in the following prefix form:

```
<function name> <argument-1> ... <argument-n>
```

Note that a function’s name and its arguments are separated only by spaces. There is no other punctuation, such as commas or parentheses.

A function application has the highest precedence in any expression, so you must be careful to enclose arguments within parentheses when they are not simple items like literal data values or variables. For example, here are two attempts evaluate an expression to compute the hypotenuse of a right triangle of sides 3 and 4:

```
Prelude> sqrt (3 ^ 2 + 4 ^ 2)
    5.0
Prelude> sqrt 3 ^ 2 + 4 ^ 2
```
19.0

Note that the first attempt produces the correct value, because the sum of the squares of the two other sides is calculated before the `sqrt` function is applied. The second attempt is incorrect, because the `sqrt` function is first applied to 3, which is then squared back to 3.0, which is then added to the square of 4, or 16, to produce 19.0.

For now, there is not much more to learn about function applications, except that they run some hidden code on their arguments and return the resulting value, and that the same arguments always produce the same value. Therefore, the only things you need to worry about are providing the function with appropriate arguments and using its returned value appropriately. The arguments must be

- of the right number (no more and no less)
- of the right types (such as, no string where an integer is expected)
- in the right positions

When you fail to adhere to these rules, the Haskell compiler will give you an error message.

How can you find out the rules or requirements for using a function like `sqrt`? As you saw in the last section, the GHCi `:type` command returns the type signature of a function or operator:

```
Prelude> :type sqrt
sqrt :: Floating a => a -> a
```

The type signature of the `sqrt` function contains a type class named `Floating` and a type variable named `a`. This notation basically says that the function takes a value of type `a` (the function’s argument) to a value of type `a` (the function’s returned value), as long as both types are members of the `Floating` type class. Examples of type `a` are `Float`, `Double`, `Integer`, and `Int`, among others. These types all support the operations required by the `sqrt` function to do its work.

You can also find documentation on the requirements of all of the Haskell functions at


The built-in Haskell functions are organized by topic in libraries. When you link to this page, scroll down to the `Prelude` library and take this link. The Haskell `Prelude` includes functions that are automatically available whenever you start the GHCi. For example, if you select `Numeric Functions` in the `Contents` section of the `Prelude` page, you will see the type signatures of such functions as `even`, `odd`, and `gcd` (short for greatest common divisor). Run these functions in the GHCi to see what they do, and then think about the types in the functions’ type signatures. Finally, try this also with some of the trigonometric functions, such as `sin`, `cos`, and `tan`. Most of the time, the `:type`
command in the GHCi will give you enough information to be able to use a function correctly.

Many other Haskell functions can be imported from standard libraries, also called modules, a topic to which you will now turn.

**Modules**

A *module* or *library* provides a place where related functions are defined and accessed. The Haskell Platform comes with dozens of modules that include functions for almost any occasion. The Haskell *Prelude* module, also called the Standard Prelude, includes functions that are always available. The functions in other Haskell modules must be imported into the GHCi or into your modules before they can be used. Your standalone applications are also modules; you have seen a couple of these already in this book. In this section, you will learn how to import items from existing modules. In the next section, you will learn how to organize your own functions in modules.

If you take the link to the Haskell libraries, you will notice that the modules are organized in a hierarchy. When you scroll down to the *Data* module, you will see nested under it some sub-modules, such as *Data.Array*, *Data.Bool*, and *Data.Char*. If you take the link to *Data.Char*, you will see some library functions for Haskell’s *Char* data type; these functions are useful for text processing. Many of them take a character as an argument and test it for a certain property. For example, the *isUpper* function returns *True* if its argument is an uppercase letter, or *False* otherwise. Other functions transform a character in some way, such as *digitToInt*, which converts a digit to the corresponding integer value.

To try out some of these functions, you import the module and then run the functions in GHCi. Here are some examples:

```
Prelude> import Data.Char
Prelude Data.Char> isLetter('K')
True
Prelude Data.Char> isLower('K')
False
Prelude Data.Char> digitToInt('9')
9
Prelude Data.Char> ord('9')
57
Prelude Data.Char> chr(57)
'9'
```

Note the difference between the *digitToInt* function and the *ord* function. The latter function gives the digit’s ASCII value, which is not the same as the integer that the digit represents. Also, note that the *ord* and *chr* functions are inverses of each other.

The *import* operation used in the last example is called a *full import*, because it makes all of the items defined in that module available to your code. Occasionally, you will want to be more selective, because you might want to use a name in your code that conflicts with a name from an imported module. There are several ways to accomplish this.
First, when you want to use a module’s functions whose names don’t conflict with names in your own code, you can perform a *partial import*. To do this, you list the desired functions in parentheses after the module’s name. For example, the `import` expression in the previous code segment can be converted to the following one:

```
Prelude> import Data.Char (isLetter, isLower, digitToInt, ord)
```

Now you can use each of these four functions, just as you did in the previous code segment, but you cannot access any of the other names in the module, so they won’t conflict with your code’s names. Another advantage of a partial import is that it tells the reader (and the Haskell compiler) exactly which module functions you will be using.

Second, when you must use a module’s function that does conflict with a name in your own code, you can perform a *qualified import*, either of the entire module or of an explicit set of items. The next example repeats the earlier one using a qualified import of the entire module:

```
Prelude> import qualified Data.Char
Prelude Data.Char> Data.Char.isLetter('K')
True
Prelude Data.Char> Data.Char.isLower('K')
False
Prelude Data.Char> Data.Char.digitToInt('9')
9
Prelude Data.Char> Data.Char.ord('9')
57
Prelude Data.Char> Data.Char.chr(57)
'9'
```

Note that you must now use the module name as a prefix for each function in the your code. Some programmers actually believe that qualified imports make their code more readable; when name conflicts are not an issue, using qualified imports is probably a matter of individual taste.

Third and finally, when you want to use a name that conflicts with a name from another module and you’re not going to use the name as defined in the latter module, you can import that module by *hiding* the unused name. This is most often done when you want to override the definition of a variable or function that’s included in Haskell’s *Prelude* module. For example, if you want to use a less precise definition of `pi` than the one provided by the *Prelude*, you can do it as follows:

```
Prelude> import Prelude hiding (pi)
Prelude> let pi = 3.14
```

You actually don’t need this type of import to override the definitions of built-in names in the GHCi, but your own modules will require it.

In the next section, you will learn how to define functions in a module and export them therefrom.
Defining Simple Functions

Although Haskell comes with hundreds of built-in functions, you will have to include them in more complex expressions to solve interesting problems. Consider, for example, the problem of computing the hypotenuse of a right triangle, as discussed in the Introduction to this book. An expression to compute a right triangle’s hypotenuse with the sides 3 and 4 was shown earlier:

```
Prelude> sqrt (3 ^ 2 + 4 ^ 2)
5.0
```

While this expression is correct, there are several problems with it:

1. You have to key in the entire thing and get it right, every time you want to calculate a right triangle’s hypotenuse.
2. If you just want to obtain the hypotenuse, given the other two sides of a right triangle, you will have to think about Pythagoras and geometry every time you do this.
3. If you discover a better way to compute this result, you will have to track down every place in your code where you do it to update your software.

To use another example, programmers who want the square root of a number do not have to think about or write the complex calculations required for that; they just call the `sqrt` function with the given number as an argument. If a better method of computing square roots comes along, the folks who maintain the Haskell Platform change the definition of the `sqrt` function in the Standard Prelude. Then, the thousands of Haskell programmers who use this function just upgrade their compilers and recompile their code.

It would be convenient to have at your disposal a `hypotenuse` function, which takes the other sides of a right triangle as arguments and returns its hypotenuse. Programming is the discipline of creating such conveniences.

To do this for the hypotenuse problem, you need to generalize the formula to work for any two sides, called `a` and `b`, as follows:

```
sqrt (a ^ 2 + b ^ 2)
```

Then you use this formula in an equation whose left side specifies the name of the function and its arguments:

```
hypotenuse a b = sqrt (a ^ 2 + b ^ 2)
```

The form of a simple function definition is

```
<function name> <parameters> = <expression>
```

where the function name has the same form as a variable name, the parameters are names separated by spaces, and the expression is any expression.

In the context of the GHCi, you must embed this definition in a `let` expression; then you can try it out with different arguments or look up its type:
Prelude> let hypotenuse a b = sqrt (a ^ 2 + b ^ 2)
Prelude> hypotenuse 3 4
5.0
Prelude> hypotenuse 9 16
18.35755975068582
Prelude> :type hypotenuse
hypotenuse :: Floating a => a -> a

Note the flowing points about this session:

- The use of `let` to define a function looks almost the same as its use to give a name to a variable. In the case of a function definition, you give a name to a piece of code that can be run, along with some names for the values (arguments) that will be used during any particular execution.
- The names of the arguments to the left of the `=` sign, also called parameters, receive their values each time the function is applied to its arguments. These values are then substituted for the parameters throughout the code on the right side of the definition. You will take a closer look at this substitution process later in the book.
- Like variables, functions have type signatures. The type signature of the `hypotenuse` function includes the types of its arguments and its return type. In this example, the GHCi infers the `hypotenuse` function’s type from the use of the `sqrt` function and the `^` operator in the definition.

When created with a `let` expression in the GHCi, a function’s definition must occupy only a single line of code. This is fine for trying out simple function definitions interactively, but you will soon need to create longer definitions, organize them by topic, and save them permanently in files. For all of these reasons, you will need to put your function definitions in modules.

Now that you have a `hypotenuse` function that works correctly, you can open your favorite text editor and add the function to a new module named `Geometry`. Here is the code for this module:

```haskell
{-
File: Geometry.hs
A module of functions for working with geometric shapes
-}

module Geometry (hypotenuse) where

-- Computes and returns the hypotenuse of a right triangle
-- with sides a and b.
hypotenuse :: Floating a => a -> a -> a
hypotenuse a b = sqrt (a ^ 2 + b ^ 2)
```

The module begins with a prefatory comment (the text enclosed between `{` and `}`).
This comment describes the module’s purpose.

The prefatory comment is followed by the module’s heading, which includes a parenthesized list of function names to be exported to other modules. If this list is
omitted, then all of the module’s names are exported; but it’s considered good programming practice to explicitly mention your exports.

Following the module heading, you see some end of line comments (two lines beginning with --) that describe what the hypotenuse function does.

Finally, there appear the function’s type signature and its definition, on separate lines. The type signature is not required, but aids the human reader and the Haskell compiler. The function’s definition is not embedded in a let expression, as it must be in the GHCi.

When you have saved the Geometry module in the file Geometry.hs, you can load it into the GHCi as usual, and try out your new library function:

```haskell
Prelude> :load Geometry
[1 of 1] Compiling Geometry             ( Geometry.hs, interpreted )
Ok, modules loaded: Geometry.
Geometry> hypotenuse 3 4
5.0
```

Variable definitions and function definitions can occur in the same module. Variable definitions look like (and actually are) definitions of functions with no arguments. Here is a trivial example:

```haskell
--File: Trivial.hs

module Trivial where

aFunction :: Int
aFunction x = x * aVariable  -- Uses a module variable

aVariable :: Int
aVariable = 2
```

Note that a type signature always precedes each definition of a variable or function. However, function definitions can appear in any order within a module.

Exercises
1. Use a text editor to create a module named Circle, in a file named Circle.hs. Define and export the functions diameter, circumference, area, surfaceArea, and volume. Then load the module into the GHCi and test your functions.

2. Use a text editor to create a module named Conversions, in a file named Conversions.hs. Define and export the functions fahrenheitToCelsius, celsiusToFahrenheit, feetToMeters, metersToFeet, and milesToKilometers, and kilometersToMiles. Each function expects one argument and returns the appropriate value. Then load the module into the GHCi and test your functions.

Recursive Functions and Pattern Matching
The Haskell expressions that you have seen thus far, even when they are complex, involve simple calculations. Other kinds of computation require repeating a given set of operations. Some of these operations, such as reading data values from a file, must be repeated until a condition (the end of file) becomes true. Other operations, such as
calculating the sum of a range of integers, require repeating an operation a fixed number of times. In this section, you explore how to define recursive functions, which perform both kinds of repetition.

In non-functional language like Java or Python, you use a loop construct to repeat an operation or set of operations. The most commonly used loop construct has the following form, in pseudocode:

```pseudocode
while the loop must continue
  perform operation-1
  ...
  perform operation-n
```

This type of loop checks a condition at loop entry. If the condition is true, the loop runs the sequence of operations in the loop’s body and returns to check the condition once more. Otherwise, if the condition is false, the loop terminates and control moves to the code below the loop construct. An operation in the loop body typically updates a variable, which eventually causes the loop control condition to become false, thereby stopping the iterative process.

For example, the following pseudocode loop computes the factorial of 4, which is the product of the numbers between 1 and 4, inclusive:

```pseudocode
product = 1
n = 4
while n > 0
  product = product * n
  n = n - 1
```

In this code, the value of `product` gets larger and the value of `n` gets smaller, until `n` becomes 0 and the loop halts. At that point, `product` is the factorial of `n`.

This way of describing a process that repeats itself is not possible in a functional language like Haskell, for three reasons:

1. No loop constructs are available.
2. You cannot make multiple assignments to the same variable; variables are single-assignment only.
3. You cannot write a sequence of operations; the only way to schedule operations is by composing expressions.

But all is not lost. In a functional language like Haskell, you use a recursive function to repeat an operation. A recursive function repeatedly calls itself, until a special condition, called the base case, is reached.

The easiest recursive functions to develop come from recursive definitions of functions in mathematics. Moreover, the code for these functions is often simpler than the loop constructs used in non-functional languages. For example, consider the factorial of an integer `n`, written as `n!`, which is the product of the integers 1 through `n`. Here is a recursive definition of `n!`: 
\[ n! = 1, \text{ when } n = 1 \]
\[ n! = n \times (n - 1)!, \text{ when } n > 1 \]

As you can see, the first part of this definition is the base case, and the second part involves a recursive step, where the ! symbol appears on both sides of the equation.

In Haskell, the definition of a function that computes \( n! \) includes two equations that express these possibilities. Here is the code for these equations, also called clauses, along with a type signature:

\[
factorial :: \text{Integer} \rightarrow \text{Integer} \\
factorial 1 = 1 \\
factorial n = n \times \text{factorial} \ (n - 1)
\]

Going from the mathematical definition to the Haskell function definition is straightforward, but how will this function, which has two clauses and calls itself in one of them, work when it is called with an argument?

Note that the first clause of the \( \text{factorial} \) function does not have a named parameter. Instead, this parameter is a literal value of 1. When the function is called, the runtime system attempts to match the function’s argument to the 1 in the first clause. If the argument is 1, the match to the first clause succeeds, and the value 1 is returned. This of course is the base case of the recursive process.

Any other argument (that is, any integer greater than 1) causes the match to fail, so the runtime system attempts to match the argument to the parameter \( n \) of the second clause. Because this parameter is a variable, this match will always succeed. In that case, the argument is substituted for the variable \( n \) throughout the code on the right side of the second clause, and the computation continues.

Here is a trace of the recursive computations that begins with a top-level call of \( \text{factorial} 3 \):

\[
\text{factorial} 3 \rightarrow \\
3 \times \text{factorial} 2 \rightarrow \\
2 \times \text{factorial} 1 \rightarrow \\
\quad \quad \quad \rightarrow 1 \\
\quad \quad \quad \rightarrow 2 \\
\quad \quad \rightarrow 6
\]

When second clause of the function is chosen, the argument becomes smaller by one and the recursive process moves forward. These steps continue until the argument becomes 1. When that happens, the first clause is chosen, the function’s value is returned, and the recursive process begins to unwind. Note that the values of earlier arguments have been remembered, so they can be multiplied by the values returned by each recursive call of \( \text{factorial} \).

Sometimes a recursive process will have more than one base case. Consider the Fibonacci sequence, which begins with the integers 1 and 1 and thereafter includes integers that are the sums of the previous two integers:
Here is a recursive definition of the \( n \)th Fibonacci number in this sequence:

\[
\text{fib}(n) = 1, \text{ when } n = 1 \text{ or } n = 2
\]

\[
\text{fib}(n) = \text{fib} (n - 2) + \text{fib} (n - 1), \text{ otherwise}
\]

Since there are two base cases, when \( n = 1 \) or when \( n = 2 \), the Haskell definition of the \texttt{fib} function includes three clauses:

\[
\textbf{fib} :: \textbf{Integer} \rightarrow \textbf{Integer}
\]

\[
\text{fib} 1 = 1
\]

\[
\text{fib} 2 = 1
\]

\[
\text{fib} n = \text{fib} (n - 2) + \text{fib} (n - 1)
\]

A trace of this function with a given argument is left for you as an exercise.

As you can see from the examples of the \texttt{factorial} and \texttt{fib} functions, the move from the mathematical definition of a given function to a Haskell function definition can be fairly mechanical. Haskell’s mechanism of pattern matching, whereby a function’s argument matches either a literal value or a named parameter, makes the design of the appropriate function clauses straightforward. However, not all recursive definitions have literal base cases, and some repetitive processes require more general termination conditions. You will explore examples of these problems and their solutions shortly.

**Exercises**

1. Recall Haskell’s \(^\texttt{^} \) operator for exponentiation, which raises a base to a given exponent. Here is a recursive definition of exponentiation for non-negative exponents:

\[
\text{base}^0 = 1
\]

\[
\text{base}^n = \text{base} \times \text{base}^{n - 1}, \text{ otherwise}
\]

Use a text editor to create a module named \texttt{MyMath}, in a file named \texttt{MyMath.hs}. Define and export the function \texttt{pow}. This function expects a base and a non-negative exponent as arguments and returns \texttt{base}^{\text{exponent}} (do not use the \(^\texttt{^} \) operator for this!). Load the module into the GHCI and test your function.

2. Draw a trace of the calls of the \texttt{fib} function, starting with \texttt{fib(5)}. The trace should show a tree-like diagram branching down from the top-level call, with two recursive calls below each call. Represent each call as a circle enclosing its argument.

3. A standard science experiment is to drop a ball and see how high it bounces. Once the "bounciness" of the ball has been determined, the ratio gives a bounciness index. For example, if a ball dropped from a height of 10 feet bounces 6 feet high, the index is 0.6 and the total distance traveled by the ball is 16 feet after one bounce. If the ball were to continue bouncing, the distance after two bounces would be 10 ft + 6 ft + 6 ft + 3.6 ft = 25.6 ft. Note that distance traveled for each successive bounce is the
distance to the floor plus 0.6 of that distance as the ball comes back up. Define a function `bouncy` that expects as arguments the initial height of the ball, its bounciness index, and the number of times the ball is allowed to continue bouncing. The function returns the total distance traveled by the ball.

**Functions with Guards**
The recursive functions discussed in the last two sections use pattern matching to select one of several possible function clauses. Unfortunately, this mechanism does not handle cases where the options depend on other conditions. For example, consider the functions `max` and `min`, which return the larger or smaller of their two arguments, respectively:

```haskell
Prelude> max 2 3
3
Prelude> min 2 3
2
```

There is no pattern to match here; the two arguments must be compared and the appropriate result returned, based upon the outcome of this test. Haskell provides a feature called a _guard_ for this purpose. Here is the form for its use:

```haskell
<function name> <argument-1> … <argument-n>
  | <Boolean expression-1> = <expression-1>
  …
  | <Boolean expression-n> = <expression-n>
  | otherwise = <default-expression>
```

You should think of this notation as a set of function clauses preceded by conditions, where each condition guards a clause. Conditions are evaluated in sequence until one of them returns `True`, or the `otherwise` case is reached. When a condition returns `True`, the value of the expression on the right side of the guarded clause is returned. The `otherwise` case ensures a default value for the options if none of the conditions succeed. Note that indentation again is significant in this code.

Armed with the guard notation, you can write your own definition of the maximum of two integers, as follows:

```haskell
myMax :: Integer -> Integer -> Bool
myMax x y
  | x > y = x
  | otherwise = y
```

As written, this definition must appear in a module, but you can also get away with the following definition of `myMax` in the GHCi:

```haskell
Prelude> let myMax x y | x > y = x | otherwise = y
Prelude> :type myMax
Ord a => a -> a -> a
```

Although this definition is a bit harder to read, it has the advantage that, thanks to the GHCi’s type inference, the function can be applied not just to any two integers, but also
to any two values that belong to the same ordinal data type, such as floating-point numbers or strings. When you define this function in a module, you can generalize its type by including the type signature

```
Ord a => a -> a -> a
```

You will learn more about type generalization shortly.

Guards are used not just in simple functions like `max` and `min`, but also in recursive functions. For example, consider a function named `summation`, which returns the sum of the integers between a lower bound and an upper bound:

```
Prelude> summation 4 4
4
Prelude> summation 3 4
7
Prelude> summation 2 4
9
Prelude> :type summation
Integer -> Integer -> Integer
```

As you can see, the function returns the lower bound when the lower bound and the upper bound are equal. Otherwise, function appears to return the sum of the lower bound and the summation of next lower bound and the upper bound. The following recursive definition captures this informal description of the summation function:

```
summation(lower, upper) = lower, when lower = upper

summation(lower, upper) = lower + summation(lower + 1, upper), otherwise.
```

Because the base case must compare the lower and upper bounds, you use guarded clauses in a Haskell `summation` function, as follows:

```
summation :: Integer -> Integer -> Integer
summation lower upper
  | lower == upper = lower
  | otherwise = lower + summation (lower + 1) upper
```

Here is a trace of the call of `summation 2 4`:

```
summation 2 4 -> -- lower < upper
  2 + summation 3 4 -> -- lower < upper
    3 + summation 4 4 -> -- lower == upper
      <- 4 -- Returns 4 (lower)
    <- 7 -- Returns 3 + 4
    <- 9 -- Returns 2 + 7
```

**Exercises**

1. In the 17th century, Sir Isaac Newton discovered an algorithm for approximating the square root of a positive number. Recall that the square root \( y \) of a positive number \( x \) is the number \( y \) such that \( y^2 = x \). Newton discovered that if one’s initial estimate of \( y \) is \( z \), then a better estimate of \( y \) can be obtained by taking the average of \( z \) together with \( x/z \). The estimate can be transformed by this rule again and again, until a
satisfactory estimate is reached. Define a function named `newton`, which takes three numbers as arguments. The first argument is a positive number \( x \), the second argument is the estimate of the square root of \( x \), and the third argument is a tolerance value. The function uses Newton’s method to improve the estimate of the square root of \( x \) until this estimate is satisfactory. It returns the estimate when the difference between its square and \( x \) becomes less than the tolerance value.

2. Restructure your code for the `newton` function of Exercise 1 with the helper functions `goodEnough` and `improve`. The Boolean function `goodEnough` determines whether or not the `newton` function’s current estimate of the square root of \( x \) is within the tolerance value. The `improve` function returns an improved estimate of the square root of \( x \), given the `newton` function’s current estimate and \( x \).
if/then/else and case Expressions

The guard mechanism suffices for selecting an option in function definitions, but is not available for making choices in the context of other constructs, such as let expressions. Like other languages, Haskell provides a more general control structure for making two-way decisions, called the if/else expression. Here is its form:

if <Boolean expression> then <expression-1> else <expression-2>

The entire if/else expression returns the value of one of the two alternative expressions following then or else: the first expression if the condition is True, or the second expression if the condition is False. You can try out this type of expression in the GHCi, as follows:

Prelude> let x = 2
Prelude> let y = 3
Prelude> if x > y then x else y
3

Note that this appears to be the same logic that you used earlier to define the myMax function, but is represented by other means. You can easily redefine the myMax function if you prefer the if/else expression to the guards:

let myMax x y = if x > y then x else y

Some programmers argue that the use of the if/else expression can be cleaner and simpler than the use of guards. When used in more complex expressions in modules, the if/else expression is best written on multiple lines, but then spacing and indentation become important. Here are revised versions of the myMax and factorial functions using this format:

myMax x y =
    if x > y then
        x
    else
        y

factorial n =
    if n == 1 then
        1
    else
        n * factorial (n - 1)

Note how the line breaks and indentation highlight the condition and the two alternative expressions in each definition.

The if/else expression can be nested to handle multiple conditions and options. For example, consider an old-fashioned, text-based command interpreter, where the user enters a letter to indicate a command. In this example, the letters N, O, S, and Q mean the commands New File, Open File, Save File, and Quit Program, respectively. The function interpretCommand expects a letter as an argument and calls another function to
run the corresponding command. The user is allowed to enter either lowercase or uppercase letters, so the interpreter must convert its argument to an uppercase letter before making its choice. You assume that the functions `newFile`, `openFile`, `SaveFile`, and `quitProgram` have already been defined and return a string (you will explore the `String` data type shortly). Here is the code for this function, which uses a series of nested `if/else` expressions to select a command:

```haskell
module Interpreter (interpretCommand) where

import Data.Char (toUpper)

interpretCommand :: Char -> String
interpretCommand letter =
  if toUpper letter == 'N' then
    newFile
  else if toUpper letter == 'O' then
    openFile
  else if toUpper letter == 'S' then
    saveFile
  else
    quitProgram

newFile, openFile, SaveFile, quitProgram :: String
newFile = "New File"
openFile = "Open File"
saveFile = "Save File"
quitProgram = "Quit Program"

After loading this module in the GHCi, you can test the function like this:

```
Interpreter> interpretCommand 'N'
"New File"
Interpreter> interpretCommand 'Q'
"Quit Program"
```

This function works correctly, but expressing all of the conditions with nested `if/else` expressions is rather cumbersome. A simpler control structure to use in this situation is the `case` expression. Here is its form:

```haskell
case <selector expression> of
  <value-1> -> <expression-1>
  ..
  <value-n> -> <expression-n>
```

The selector expression must evaluate to one of the values listed below the `case` expression’s heading, to the left of the `->` symbol. The `case` expression then returns the value of the corresponding expression to the right of the `->` symbol. Here is the code for a simplified command interpreter:

```haskell
import Data.Char (toUpper)

interpretCommand :: Char -> String
```
interpretCommand letter =
    case toUpper letter of
        'N' -> newFile
        'O' -> openFile
        'S' -> saveFile
        'Q' -> quitProgram
        _  -> handleError

handleError = "Error: illegal command!"

In the second version, you can add a final clause to the case expression that begins with the _ symbol, which matches any value in Haskell:

interpretCommand :: Char -> String
interpretCommand letter =
    case toUpper letter of
        'N' -> newFile
        'O' -> openFile
        'S' -> saveFile
        'Q' -> quitProgram
        _  -> handleError

Exercises
Redefine several of the functions mentioned earlier (fib, summation, and pow) so that they use an if/else expression instead of function clauses. State which version you prefer and explain why.
Tail Recursion
The recursive functions discussed so far are counterparts of simple and elegant mathematical definitions. However, their performance at runtime can lead one to doubt their usefulness. To understand why, you need to take a brief detour to consider the cost of running functions on a real computer. There are two costs: running time and memory.

A useful way to measure the running time of a recursive function is to determine the rate of growth of the number of recursive calls as a function of the size of an input argument. For example, it is easy to see from a trace of the factorial function that it requires \( n - 1 \) recursive calls. A loop-based version of this function, if you had one at your disposal, would fare no better: it would require \( n - 1 \) iterations to compute the factorial of \( n \). Thus, you can say that the rate of growth of the running time of this function is linear.

Likewise, you can compare the rate of growth of memory for both versions of factorial. The loop-based version would require memory cells for the values of \( n \) and the running total, as well as whatever information is needed to return to the caller’s context. The values of \( n \) and the running total are simply reset on each pass through the loop, and the function is called exactly once. Thus, you can say the rate of growth of memory for this version of the function is constant (that is, the memory usage does not grow with the value of \( n \)).

The rate of growth of memory for the recursive version of factorial is quite different, however. In this version, the expression \( n \times \text{factorial} \ (n - 1) \) multiplies the result of a recursive call by \( n \). Thus, the runtime system must reserve memory to remember data values to be used in computations that follow the recursive calls. Each function call requires at least two cells, one for the value of \( n \) and one for that call’s return value. This information is maintained in a chunk of memory called an activation record. Each time the function is called, the runtime system creates an activation record for its information and places that record on a data structure in memory called the call stack. Figure 5 shows the state of memory after one top-level call and 2 recursive calls during the computation of factorial 3.

**FIGURE 5**
The call stack for factorial 3

```
  n 1  Activation record for factorial 1
   \downarrow
  n 2  Activation record for factorial 2
   \downarrow
  n 3  Activation record for factorial 3
```

When a call returns, its value is placed in an empty cell for its caller to process, and its activation record is removed from the call stack. As you can see, the amount of this memory grows as a linear function of \( n \) for the factorial function.
Now you might think that a linear growth of memory is no big deal for a function like \texttt{factorial}, which is unlikely to be called with an argument greater than 100. But there might be other recursive functions that process very large input values, say, in the millions, where this behavior would become unacceptable. How then can a functional language like Haskell, with no loop construct, avoid this problem? The answer lies in the GHC, the smart Haskell compiler, which translates something called \textit{tail recursion} into an iterative loop at compile time. You will now examine the characteristics of tail recursion.

Because the need for extra memory is due to the work being done after recursive calls, you can eliminate this need by structuring the recursion so that no work is done after the recursive calls. One way to accomplish this is to include an extra argument for the result and calculate it \textit{before} each call. The result thus accumulates on the way forward into the recursive process, and is returned when the process reaches the base case. The final result is then passed all the way up the call chain, without further computations. Because the recursive call is the last thing done, this type of function is called \textit{tail recursive}.

A tail recursive version of the \texttt{factorial} function calls a tail recursive helper function named \texttt{tailFactorial}, with an initial \texttt{result} argument of 1. This function then does all of the work of computing the factorial of \textit{n}, but more efficiently than the previous version. Here is the code for the two functions:

\begin{verbatim}
factorial :: Integer -> Integer
factorial n = tailFactorial n 1

tailFactorial :: Integer -> Integer -> Integer
tailFactorial 1 result = result
tailFactorial n result = tailFactorial (n - 1) (n * result)
\end{verbatim}

Assume for the moment that the compiler does not generate a loop from this code. Here is what a trace of the recursive computations that begin with a top-level call of \texttt{tailFactorial 3 1} would look like:

\begin{verbatim}
tailFactorial 3 1-> -- Matches tailFactorial n result
   tailFactorial 2 3-> -- Matches tailFactorial n result
      tailFactorial 1 6 -> -- Matches tailFactorial 1 result
         <- 6 -- Returns 6
         <- 6 -- Returns 6
         <- 6 -- Returns 6
\end{verbatim}

Computation is achieved as the arguments are calculated before each recursive call. As you can see, once a result is returned, it never changes. Thus, when the GHC translates a tail recursive version to machine code, there will actually be a single call of the function, followed by a loop that uses a fixed amount of memory at run time.

You can upgrade the \texttt{summation} function in a similar manner. Using the variable \texttt{result} for the running total and the variables \texttt{lower} and \texttt{upper} for the bounds of the range of numbers, a pseudocode loop describes the summation as follows:
result = lower
while lower < upper
    result = result + lower
    lower = lower + 1

A tail-recursive strategy for the summation repeatedly calls the function, until the base case is reached. As before, this condition occurs when the lower bound equals the upper bound. But in this case, the final value of the running total is simply returned. Otherwise, the function

- adds the lower bound to the running total
- increments the lower bound by one
- is recursively called with these two values as arguments

The running total is initially the lower bound.

Here is the code for the definition of the summation function, which uses a tail recursive helper named tailSum:

```haskell
summation :: Integer -> Integer -> Integer
summation lower upper = tailSum lower lower

tailSum :: Integer -> Integer -> Integer -> Integer
tailSum lower upper result |
    lower == upper = result
    otherwise = tailSum (lower + 1) upper (lower + result)
```

Developing a tail-recursive fib function is a bit trickier, because there are results from two recursive calls that must be accounted for. However, you can use the insight that with the exception of the first two Fibonacci numbers, the \(n\)th Fibonacci number is the sum of the previous two Fibonacci numbers. If these two numbers are \(a\) and \(b\) and they start at 1 and 0, respectively, each Fibonacci number after the first one can be obtained by replacing \(a\) with \(a + b\) and \(b\) with \(a\). At the end of \(n - 1\) such replacements, \(a\) will be the \(n\)th Fibonacci number. There is now a single termination condition, when \(n = 1\). Here is a pseudocode loop that describes this process:

```haskell
a = 1
b = 0
while n > 1
    a, b = a + b, a -- Simultaneous assignments
    n = n - 1
```

And here is the strategy for a tail recursive process:

1. Pass three arguments, \(a\), \(b\), and \(n\), to a tail recursive helper function. The initial values of \(a\) and \(b\) are 1 and 0, so \(a\) is initially the first Fibonacci number.
2. When \(n\) reaches 1, return \(a\).
3. Replace \(a\) with \(a + b\), replace \(b\) with \(a\), and decrement \(n\) before each recursive call of the helper function.

Here is the code for the tail recursive Fibonacci function:
fib :: Integer -> Integer
fib n = tailFib 1 0 n

tailFib :: Integer -> Integer -> Integer
tailFib a b 1 = a
tailFib a b n = tailFib (a + b) a (n - 1)

As you can see, all of the computations are done when the arguments to tailFib are evaluated; no work is done after each recursive call. Thus, the value of a returned at the bottom of the recursive process never changes, so the function requires no growth of memory.

The use of tail recursion also improves the running time of this version of the fib function. As you might have noticed when you drew a diagram of the trace of the calls for fib 5 in an earlier exercise, the first version of the fib function showed an exponential rate of growth of these calls. By contrast, the tail recursive version starts with n and decrements it on each call, counting down to 1, for a total of approximately n recursive calls. Thus, the running time of the tail recursive fib function has a linear growth rate. Figure 6 shows the graphs of the rate of growth of the running time for the two versions of the fib function.

**FIGURE 6**
Rates of growth running time for two versions of the fib function

Whenever a recursive function is running too slowly or using too much memory, it is very important to find a tail recursive version.

**Exercises**
1. Convert the pow function from the earlier set of exercises to a tail recursive version. This function should call a helper function that expects a running total as an argument.
2. A faster method of exponentiation distinguishes between odd and even exponents, according to the following recursive definition:

\[
\begin{align*}
base^0 &= 1 \\
base^n &= base * base^{n-1}, \text{ if } n \text{ is odd}
\end{align*}
\]
base^n = (base^{n/2})^2 if n is even

Add the function fastPow to your MyMath module. Like the pow function from the earlier set of exercises, this function decrements the exponent by one, when the exponent is odd. But it improves the performance of pow by returning the square of the recursive call of the function with the exponent divided by 2 (integer quotient), when the exponent is even. Note: this function is not tail recursive.

3. Draw a graph that plots the number of recursive calls of pow and fastPow for n = 2, 4, 8, 16, and 32. Compare the performance of the pow and fastPow functions. How many recursive calls of each function will there be, as the exponent gets large?

where Clauses
In the previous section, you defined functions that make use of closely related helper functions. There is a way in which you can make the code for these functions safer and more readable. In these examples, the tailFactorial, tailSum, and tailFib functions are top-level functions. This means that any programmer who imports the entire module in which they are defined will be able to call any of these functions, as well as the factorial, summation, and fib functions that use them. But the helper functions are intended for the use of the functions that they help only; no one else, even within the defining module, should be able to use them or know about their existence.

To solve this problem, you can place each helper function’s definition within the definition of its client function, by using a where clause. Here is an example use of a where clause in the tail-recursive factorial function:

factorial :: Integer -> Integer
factorial n = tailFactorial n 1 where
  tailFactorial :: Integer -> Integer -> Integer -> Integer
  tailFactorial 1 result = result
  tailFactorial n result = tailFactorial (n - 1) (n * result)

In this new version, you nest the entire definition and type signature of the helper function tailFactorial within a where clause. The lines of code for the helper function tailFactorial are indented on the lines below where, to assist both the reader and the compiler. The tailFactorial function is now hidden from the rest of the module and its clients, but is still available to the factorial function.

Now let’s do the same thing for the summation function:

summation :: Integer -> Integer -> Integer
summation lower upper = tailSum lower lower where
  tailSum :: Integer -> Integer -> Integer
  tailSum lower result
      | lower == upper = result
      | otherwise = tailSum (lower + 1) (lower + result)

Note something that appears odd about this definition: the tailSum function no longer needs a third parameter to track the value of the upper bound. The reason for this is that

This is possible because of the where clause.
the value of the `upper` parameter of the `summation` function never changes, and this parameter is still visible in the code within the `where` clause.

**Exercises**  
Modify the tail-recursive version of the `fib` function so that the `tailFib` function is nested in a `where` clause.

**Dealing with Errors**  
Part I has introduced the basic types of computations available in Haskell. You have learned not only these, but also some control structures for composing more complex computations from them, and how to define functions to hide this complexity. In Part II, you will learn how to use Haskell’s built-in data structures to organize and simplify the processing of complex combinations of data values. Before you begin Part II, a few words about dealing with errors in programs are in order.

Complex programs rarely run correctly the first time they are tested, and even after extensive testing, release to the world, and years of service, users of software often discover and report errors. In this section, you’ll learn about various types of errors, how to correct them when they are discovered, and how to prevent them from happening.

There are three types of errors that can occur in software: *syntax errors*, *semantic errors*, and *logic errors*.

**Syntax Errors**  
You have probably seen some syntax errors already, especially when you mistype the name of a variable, forget to include a set of parentheses in an expression, or supply the wrong type of argument to a function application.

The good news is that the Haskell compiler catches all syntax errors before the code runs. This allows you to view an error message, track down the source of the problem, and fix it. The bad news is that syntax errors can be annoying, until you become competent enough as a programmer to avoid them. Beginning Haskell programmers often complain that parts of the language, such as static type checking, seem very strict when compared with other languages they might have learned. However, the point of such constraints is to catch as many errors as possible at compile time, thus avoiding the exhaustive testing required to do so or the embarrassment of users finding them when your code is released to the world.

**Semantic Errors**  
Semantic errors can remain in a program, even though your code has compiled without syntax errors. Semantic errors occur at run time and cause execution to halt with an error message. The term “semantic error” indicates that your code, although syntactically correct, attempts to perform a meaningless operation. For example, a function call might receive a 0 as an argument and then attempt use this as the divisor with the integer quotient operator.

The good news is that the runtime system will catch semantic errors, crash the program, and display an error message, which helps to show the source of the problem. The bad
news is that semantic errors are revealed only at runtime. This means that these errors show up either during testing, when you can observe them and correct them, or after the program is released to the world, when users experience a program crash.

**Logic Errors**

Logic errors also occur at runtime, but unlike semantic errors, they do not cause the program to crash with a helpful error message. Most logic errors cause a program to halt normally, but with outputs or other behavior that is not expected. For example, a tax calculator might include an incorrect formula to compute a value. The value is successfully computed and displayed, but it is not the expected or correct one for the problem at hand. The error lies not in the syntax or semantics of the code, but in the programmer’s choice of operations, data types, or design strategy. That’s why logic errors are also called *design errors*.

Unfortunately, logic errors are mainly bad news.

To begin with, they can be detected only by careful observation of program behavior during testing, or by careful inspection of the source code. Some progress has been made in automating the testing of software. However, source code inspection is still very much a matter of fallible human eyes and minds reading the code and searching for design flaws, once their symptoms have been detected. Needless to say, the likelihood of one or two logic errors in a program released to the world is much greater than the likelihood of one or two semantic errors.

When a logic error is detected, the cause of the problem can be much harder to track down than the cause of a semantic error. A semantic error causes a program crash, with an error message and perhaps a trace of function calls that can lead directly to the offending code. A logic error, such as incorrect output, requires the programmer to infer what might be the cause from many possible sources of evidence: the values of certain variables, the control logic used within a function definition, or perhaps the choice of functions used. Programming tools such as debuggers can help to automate some of this detective work, but much mental effort is still necessary to eliminate these types of errors.

The only good news about logic errors is that static type checking and the functional model of computation help to produce code designs that reduce the frequency of this type of error. Static type checking shifts the detection of many potential errors to compile time. The functional model of computation can also easily support formal proofs of the correctness of some program components.

**Using the error Function**

While the illustration of testing, debugging, and formal proof techniques is beyond the scope of this book, you can use a simple strategy for detecting many errors in your code. Haskell includes a built-in *error* function that can halt a program with an error message if the program detects an error from which it cannot recover. The form of this function application is

```
error <a string message>
```

For example, consider the *factorial* function defined earlier:
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)

This function assumes that the argument \( n \) is greater than or equal to 0. But what happens if the argument is negative? You should try running such a case in the GHCi to see what happens (*hint*: there will be lots of recursive calls, until . . .).

To prevent such behavior and halt a program with such an error, you can add a case for a negative argument to the definition, as follows:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n
    | n < 0 = error "Argument n must be >= 0"
    | otherwise = n * factorial (n - 1)
```

Now when you call `factorial` with a negative number, the system displays a semantic error message, that your argument value is not a member of the function’s domain. The programmer must correct the error in the enclosing program before releasing it to the world.
Part II. Basic Data Structures: Lists, Strings, and Tuples

In Part I, you learned how a function allows you to view a complex computation as a simple entity. These complex computations can include the selection of alternative computations as well as the repetition of computations. Programming in part consists of simplifying complex computations by providing convenient functional abstractions.

In a similar manner, a data structure is a means of organizing a complex set of data values so that they can be treated as a simple entity. Put another way, a data structure allows you to treat many data values as one data value. When combined with the appropriate functions for processing them, data structures provide convenient data abstractions and constitute another other major part of programming.

Part II introduces Haskell’s three basic data structures – the list, the string, and the tuple. Later in this book, you will use them to solve some interesting problems.

Lists
People make lists and use them for many different tasks in everyday life: for example, shopping lists, course and sports team rosters, and checkbook registers. Lists appear throughout the software world as well; common examples are playlists in music players and history lists in web browsers.

A software list provides a container that organizes data values, called items, by position. Each item in a list except the first one has a unique predecessor, and each item but the last one has a unique successor. The unique positions of items in a list are specified by integers ranging from 0 to the length of the list minus one.

Basic List Operations
A minimal set of operations on a list includes functions to

- create a list
- add items to the list
- access items in the list
- remove items from a list

Other operations might also be provided for convenience, such as

- test a list to see if it’s empty
- determine the number of items in a list
- glue two lists together to form a third list
- apply a function to each item in a list to produce a list of results
- apply a test to each item in a list to produce a list of items that pass the test
- apply a function to pairs of items in a list to boil them down to a single result

Haskell’s list type provides some basic standard functions for all of these operations, which you can use to define any other list processing functions that you need. Table 7 shows most of the functions mentioned above. The Haskell library Data.List also
includes many other operations on lists, and you are encouraged to browse the documentation for details.

<table>
<thead>
<tr>
<th>Operation</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>null list</td>
<td>Returns True if list is empty or False otherwise.</td>
</tr>
<tr>
<td>head list</td>
<td>Returns the first item (at position 0) in list.</td>
</tr>
<tr>
<td>tail list</td>
<td>Returns a list of the items after the first item in list.</td>
</tr>
<tr>
<td>length list</td>
<td>Returns the number of items in list.</td>
</tr>
<tr>
<td>item : list</td>
<td>Returns a list whose head is item and whose tail is list.</td>
</tr>
<tr>
<td>list1 ++ list2</td>
<td>Returns a list containing the items in list1 followed by the items in list2.</td>
</tr>
<tr>
<td>list !! index</td>
<td>Returns the item at position index in list. Positions are counted from 0.</td>
</tr>
</tbody>
</table>

Note that errors will be raised if you give an empty list to the head or tail operations, or if you give an index to !! that is less than 0 or greater than or equal to the length of the list.

You create a list by mentioning it as a literal value. A list literal is formed by enclosing zero or more items in square brackets, as shown in the next session:

```
Prelude> []
[]
Prelude> [1, 2, 3]
[1,2,3]
Prelude> [1..5]
[1,2,3,4,5]
```

Note that you can use the .. symbol to specify a range of values in a list.

Now let’s assume that the variable list names a list of five integers:

```
Prelude> let list = [1..5]
Prelude> list
[1,2,3,4,5]
```

Then, you can run the three basic functions null, head, and tail, to observe a property of the list or to access some items:

```
Prelude> null list
False
Prelude> head list
1
Prelude> tail list
[2,3,4,5]
```

Believe it or not, these three functions provide ¾ of the operations you need to define all of the other operations you might want on lists (the fourth, which adds an item to a list, will be discussed shortly).
Recursive List Operations

Note that the tail of a list is always another list. This implies that the list is a recursive data structure. That is, a list is either

1. empty (when null is True)
2. or it contains an item at its head, followed by another list at its tail

The recursive structure of a Haskell list provides the basis for some natural and elegant recursive list processing functions. For example, suppose that the length function is not built in. You could define a recursive function named myLength that has two cases:

1. The length of the list is 0, if the list is empty.
2. The length of the list is 1 + the length of the tail of the list, otherwise.

Here is the code for this function:

```haskell
myLength list
| null list = 0
| otherwise = 1 + myLength (tail list)
```

Note that this function advances through the list until it becomes empty. When that is true, the function returns 0. As the recursion unwinds, 1 is added to the result of each recursive call, in effect counting the number of items. The next trace shows this process for myLength [20, 30, 40]:

```
myLength [20, 30, 40] -> -- tail = [30, 40]
    1 + myLength [30, 40] -> -- tail = [40]
        1 + myLength [40] -> -- tail = []
            1 + myLength [] -> -- null [] is True
                <- 0 -- Returns 0
                    <- 1 -- Returns 0 + 1
                        <- 2 -- Returns 1 + 1
                            <- 3 -- Returns 2 + 1
```

Next, suppose that the !! operator does not exist. You could define a function named ith, which expects an index position and a list as arguments, and returns the item at the index position. This function has two cases:

1. Either the index is 0, so the item is at the head of the list.
2. Or the item is at position index - 1 in the tail of the list.

Thus, when the index is not 0, the function will advance through the list as it decrements the index. Eventually, the index becomes 0 and the function returns the head of the list. Here is the code for this function:

```haskell
ith index list
| index == 0 = head list
| otherwise = ith (index - 1) (tail list)
```
Note that the \texttt{ith} function is tail recursive, but the \texttt{myLength} function is not. Think about how to convert \texttt{myLength} to tail recursion; you’ll have an opportunity to rewrite it in the exercises.

You might be wondering by now what the types of these functions are, and what the type of a list looks like. Here are the types of some of the examples you have seen thus far:

\begin{verbatim}
Prelude> :type [1, 2, 3]
[1,2,3] :: Num t => [t]
Prelude> :type myLength
myLength :: [a] -> Integer
Prelude> :type ith
ith :: Integer -> [a] -> a
\end{verbatim}

There is nothing really strange here. The type signature of a list looks like a list that encloses the item type. In these three cases, the item type is a type variable, which indicates that it can be any type at all (in the case of the functions \texttt{myLength} and \texttt{ith}), or a type belonging to the \texttt{Num} class (in the case of the list \texttt{[1,2,3]}). In any case, all of the items in a given list must be of the same type. This type can be any type, including another list type, another data structure type, or even a function type.

\textbf{Constructing Lists and Pattern Matching with the : Operator}

The remaining basic list operation to be discussed is the binary operator symbol \texttt{:}. It expects an item and a list as its operands. This operator has two purposes, depending on its context of use:

1. When used as a constructor, it adds an item to the beginning (or head) of a list.
2. When used in pattern matching, it extracts the head and the tail of a list.

Here are some simple uses of \texttt{:} as a constructor, which adds an item to a list:

\begin{verbatim}
Prelude> let list = [1..5]
Prelude> list
[1,2,3,4,5]
Prelude> 0:list
[0,1,2,3,4,5]
Prelude> list
[1,2,3,4,5]
Prelude> 0:tail list
[0,2,3,4,5]
\end{verbatim}

Note that the \texttt{:} operator does not modify its second operand, the list. This is in keeping with Haskell’s prohibition on side effects. Instead, the operation builds and returns a new list with the new item at its head.

Armed with the \texttt{:} operator, you can define a recursive function \texttt{removeIth} that removes the \texttt{i}th item from a list. This function expects the index position of the item and the list as arguments, and returns a new list. It has two cases:

1. If the index is 0, return the tail of the list.
2. Otherwise, the item is removed at position \( \text{index} - 1 \) in the tail of the list, and the current head of the list is added to the resulting list.

Here is the code for the function:

```haskell
removeIth :: Integer -> [a] -> [a]
removeIth index list
    | index == 0 = tail list
    | otherwise = head list : removeIth (index - 1) (tail list)
```

As this recursion moves forward, items are stripped from the head of the list and the index decreases until it becomes 0. At that point, the tail of the list is returned. As the recursion unwinds, the items that were stripped off earlier, if there were any, are added back on to the list. Consider a call of \( \text{removeIth} \ 2 \ [20, 30, 40, 50] \), which locates and removes the value 40 at position 2. Here is a trace of the recursive process:

\[
\begin{align*}
\text{removeIth} & \ 2 \ [20, 30, 40, 50] \\
= & \ 20 : \text{removeIth} \ 1 \ [30, 40, 50] \\
= & \ 30 : \text{removeIth} \ 0 \ [40, 50] \\
< & \ [50] \\
< & \ [30, 50] \\
< & \ [20, 30, 50]
\end{align*}
\]

Note that the : operator is right associative, meaning that consecutive applications are evaluated from right to left:

```
Prelude> 20:30:40:[]
[20,30,40]
```

To use the : operator to extract a list’s head and tail, you match a nonempty list against a pattern of the form

\((x:xs)\)

where \(x\) is the head of the list and \(xs\) is its tail.

You can use any variable names that you want in the pattern, but the name \(x\) and \(xs\) are conventional among Haskell programmers. Here is an example:

```
Prelude> let (x:xs) = [1..5]
Prelude> x
1
Prelude> xs
[2,3,4,5]
```

The use of the : operator with pattern matching can greatly simplify the list processing functions you have defined thus far. For example, you can recast the \textit{myLength} and \textit{removeIth} functions as follows:
myLength :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs

removeIth :: Int -> [a] -> [a]
removeIth 0 (x:xs) = xs
removeIth index (x:xs) = x : removeIth (index - 1) xs

The use of pattern matching in these functions eliminates the calls of the `null`, `head`, and `tail` functions, as well as the use of comparisons and guards.

**Appending Lists with ++**

One other commonly used list operator is `++`, which places the contents of its two list arguments in a new list. Here are some examples of its use:

```
Prelude> [] ++ [2,3]
[2,3]
Prelude> [2,3] ++ []
[2,3]
Prelude> [2,3] ++ [2,3]
Prelude> [2,3,2,3]
```

From these examples, you can discern a recursive strategy for defining your own `append` function that could be used if `++` did not exist. It has two cases:

1. If the first list is empty, return the second list.
2. Otherwise, add the head of the first list to the result of appending the tail of the first list and the second list. Here is the code for this function:

```
append :: [a] -> [a] -> [a]
append [] list2 = list2
append (x:xs) list2 = x : append xs list2
```

**Sorting a List**

Many applications make use of sorted lists. For example, the chapters in a book’s table of contents are organized in ascending numeric order, and the GPAs in a senior class of students are organized in descending order. One precondition on the data in a sorted list is that the type of items must support the comparison operators. In Haskell, the way this constraint is expressed is to say that the item type must belong to the `Ord` type class (recall the type signatures of the `max` and `min` functions). The module `Data.List` includes a `sort` function that places the items in its list argument in ascending order, as shown in the following example:

```
Prelude> import Data.List (sort)
Prelude Data.List> :type sort
sort :: Ord a => [a] -> [a]
Prelude Data.List> sort [40, 20, 50, 100, 70]
[20,40,50,70,100]
```
As you can see, the `sort` function works as expected, but it can be expensive to run as the list becomes very large. One way to avoid this cost is to put items in their proper places in the list at insertion time. You can define a new function named `insert` that expects an item and a sorted list as arguments. This function’s recursive strategy has three cases:

1. The list is empty, so you wrap the new item in a list.
2. The new item is less than or equal to the item at the head of the list, so you add it to the head of the list (by placing it in front of the current head of the list).
3. Otherwise, you insert the item into the tail of the list, and add the item at the head of the list to the result.

Here is the code for the `insert` function, followed by some trial runs:

```haskell
insert :: Ord a => [a] -> a -> [a] -> [a]
insert item [] = [item]
insert item (x:xs)
    | item <= x = item : x : xs
    | otherwise = x : insert item xs
```

Prelude> insert 30 []
[30]
Prelude> insert 30 [10, 20]
[10,20,30]
Prelude> insert 20 [10, 30]
[10,20,30]

Now suppose that the `Data.List.sort` function did not exist. You could use your `insert` function to define your own sorting function named `insertionSort`. The basic idea behind an insertion sort is to sort the items in the tail of a list, and then insert the item at the list’s head into the result. The code for this surprisingly simple strategy follows:

```haskell
insertionSort :: Ord a => [a] -> [a] -> [a]
insertionSort [] = []
insertionSort (x:xs) = insert x (insertionSort xs)
```

As you can see, much of list processing is recursive, beautiful, and fun to design and code. The exercises will give you some practice!

**Exercises**

1. Define a new version of the function `myLength` that uses tail recursion.
2. Define the function `removeOne`, which expects an item and a list as arguments. This function removes the first instance of the argument item, and returns the modified list.
3. Define the function `removeAll`, which expects an item and a list as arguments. This function removes the all instances of the argument item, and returns the modified list.
4. Define the functions `replaceOne` and `replaceAll`. These functions expect two items, a target and a replacement, and a list as arguments. They replace the target item with the replacement item in the argument list, and return the modified list.
Strings
A string is a sequence of zero or more characters. Strings of characters form the basis of text processing. In this section, you will explore several types of operations on strings.

Literal strings in Haskell are written within double quotation marks. However, strings also happen to be lists of characters. Therefore, all of the list operations that you learned in the previous section, as well as any others, will work with strings. Here are some examples:

```
Prelude> let firstName = "Ken"
Prelude> :type firstName
firstName :: [Char]
Prelude> head firstName
'K'
Prelude> tail firstName
"en"
Prelude> firstName !! 2
'n'
Prelude> ['K', 'e', 'n']
"Ken"
Prelude> "" == []
True
Prelude> let lastName = "Lambert"
Prelude> firstName ++ lastName
"KenLambert"
Prelude> firstName ++ " " ++ lastName
"Ken Lambert"
```

Note that the GHCi prints a list of characters as a string. The empty string can be written either as "" or as [].

String Comparisons
Like the Char type, the string type belongs to the Ord type class. Thus, you can compare strings and arrange them in alphabetical order. For example, given a randomly ordered list of names, you can sort them by running the function `Data.List.sort`, as in the following example:

```
Prelude> import Data.List (sort)
Prelude Data.List> let names = [lastName, firstName]
Prelude Data.List> names
["Lambert","Ken"]
Prelude Data.List> "Ken" < "Lambert"
True
Prelude Data.List> sort names
["Ken","Lambert"]
```

In fact, any function whose type signature includes the type `Ord a => [a]`, such as `max`, `min`, or the `insert` function defined earlier, will work with strings.

The library `Data.String` includes functions to extract lists of words from strings and join these words back together again to form strings:
Note that the `words` function uses whitespace (blanks, newlines, or tabs) to recognize words in a string, while the `unwords` function places a single space between the words in a list to form a string. The same library includes the functions `lines` and `unlines`, which convert between strings containing line breaks (the `'\n'` character) and lists of lines of text.

Using the `show` Function
Occasionally, you will want to convert a given data value to its corresponding string representation for further text processing. For example, consider the task of numbering the lines of text in a string (perhaps for display in a text editor such as Editra). You might create a list of lines from the string using the `Data.String.lines` function, and then add line numbers to these lines of text as you build a new string from them. But each line number, which is an integer, must first be converted to a string of digits, before you can use it with the `++` operator and its line of text. Fortunately, Haskell includes a function named `show` that does this, for built-in data types that belong to the `Show` type class. Here are some example uses of the `show` function:

```haskell
Prelude> :type show
show :: Show a => a -> String
Prelude> show 34
"34"
Prelude> show 3.14
"3.14"
Prelude> show True
"True"
Prelude> show "Ken Lambert"
""Ken Lambert""
```

This function essentially wraps double quote marks around its argument, when it’s a basic data value (note that a string gets an extra pair of quotes as a result).

Armed with `show`, you can define a new function named `numberLines`, which numbers the lines within a string that contains line breaks, as follows:

```haskell
import Data.String (lines, unlines)

numberLines :: [Char] -> [Char]
numberLines text = unlines (helper (lines text) 1) where
  helper :: [[[Char]]] -> Int -> [[[Char]]]
  helper [] _ = []
  helper (x:xs) count =
    (show count ++ " " ++ x) : helper xs (count + 1)
```
Here is a run of this function; note how the newlines take effect when the string is explicitly printed with `putStrLn`:

```
Prelude> numberLines "Kenneth\nAlfred\nLambert"
"1 Kenneth\n2 Alfred\n3 Lambert\n"
Prelude> putStrLn (numberLines "Kenneth\nAlfred\nLambert")
1 Kenneth
2 Alfred
3 Lambert
```

The `numberLines` function makes use of a nested function named `helper`. The `helper` function expects a list of lines as an argument and returns a list of numbered lines. Aside from the `show` function, the only new feature in this code is the use of the `_` symbol in the first clause of the `helper` function. When used in pattern matching, this symbol matches to any value. The `_` symbol is used when you don’t care what the value is.

Somewhat surprisingly, there are no Haskell functions to convert strings to uppercase and lowercase, but it’s easy to define your own functions for these purposes. For example, consider the function `makeUppercase`, which converts the letters in a string to uppercase. The recursive strategy has a base case of the empty string (""). The recursive step extracts the character at the head of a nonempty string, uses the `Data.Char.toUpper` function to convert it to uppercase, and adds this result to the recursion on the tail of the string. Here is the code for this function:

```
import Data.Char (toUpper)
makeUppercase :: String -> String
makeUppercase "" = ""
makeUppercase (x:xs) = toUpper x : makeUppercase xs
```

Note that you can use the name `String` for the data types in the function’s type signature, rather than the type `[Char]`. The type name `String` is actually a synonym for `[Char]`. This relationship is defined in Haskell’s Prelude as follows:

```
type String = [Char]
```

**Representing Numbers as Bit Strings**

To conclude this section, consider the binary number system, which uses the digits 0 and 1 to represent unsigned integers. Such binary representations are called bit strings. Each digit in the binary system represents a given power of 2. For example, the unsigned bit string 1011 represents

\[ 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11 \]

Note that the exponents of 2 associated with each binary digit begin with the largest value at the leftmost digit and decrease by one until reaching 0 at the rightmost digit. To convert a string of bits to the corresponding unsigned integer, you can define a function named `unsignedBinaryToDecimal`. This function expects a string as an argument and returns an integer. The function uses a type synonym named `BitString` for clarity and
uses a helper function that expects a bit string, an exponent, and the running total as arguments. Here is the code for these definitions:

```haskell
type BitString = String

unsignedBinaryToDecimal :: BitString -> Int
unsignedBinaryToDecimal bitString =
    helper bitString (length bitString - 1) 0
    where
        helper :: BitString -> Int -> Int -> Int
        helper "" _ result = result
        helper (x:xs) exponent result
            | x == '0' = helper xs (exponent - 1) result
            | otherwise = helper xs (exponent - 1)
                          (result + 2 ^ exponent)
```

Note that the `helper` function moves through the digits from left to right in the bit string, and counts down through the exponents as it does so. On each recursive step, the integer result is incremented by the appropriate power of 2 only if the digit is 1.

The inverse of `unsignedBinaryToDecimal` is named `unsignedDecimalToBinary` and converts an unsigned integer to a bit string. The `helper` function is now used only when the integer is not 0. The `helper` function takes an integer and a bit string, which is initially empty, as arguments. Its strategy is to repeatedly divide the integer by 2 and prepend the remainder, as a digit, to the result string. This process continues until the integer becomes 0, when the result string is returned. Here is the code for this function:

```haskell
unsignedDecimalToBinary :: Int -> BitString
unsignedDecimalToBinary 0 = "0"
unsignedDecimalToBinary number =
    helper number ""
    where
        helper :: Int -> BitString -> BitString
        helper 0 result = result
        helper number result =
            let quotient = number `div` 2
                remainder = number `mod` 2
            in
                helper quotient (show remainder ++ result)
```

Note the use of the extended `let` expression in the `helper` function. When used in a Haskell script file, the `let` expression can take a more complex and powerful form than the one you have seen used in the GCHi:

```haskell
let <variable name-1> = <expression-1>
    <variable name-2> = <expression-2>
    . .
    <variable name-n> = <expression-n>
in <expression>
```

In this form, a `let` expression can give names to several values and then evaluate an expression in the context of these definitions. The `let` expression now returns the value
of this expression. Note that the format and use of whitespace is important: the Haskell compiler must see that each variable definition appears on its own line and begins in the same column.

**Exercises**

1. Define a function `unsignedDigitsInBaseToDecimal`, which generalizes the `unsignedBinaryToDecimal` function to converts numbers in bases 2 through 10 to decimal integers. The new function expects an integer base (2 through 10) and a string of digits in that base as arguments. It returns the unsigned integer represented by the arguments.

2. Define a function named `split`. This function behaves like the `String.words` function, which takes a string as an argument and returns a list of words in the string.

3. Define a function named `join`, which is the inverse of the `split` function.

**Tuples**

Lists and strings are linear sequences of zero or more data values of the same type. Occasionally, you will need to structure a fixed number of data items that may or may not be of the same type. For example, a student’s record in the registrar’s system might consist of a name (a string), a class year (an integer), and a GPA (a floating-point number). Haskell’s *tuple* data structure provides a convenient way to organize such data. In this section, you will explore the use of tuples to structure data.

Tuple literals look like list literals, but use parentheses instead of square brackets as delimiters. For example, suppose the information for a student consists of a name, class year, and GPA. You can create records of information for the students Bill and Mary, and add them to a list of student records, as follows:

```haskell
Prelude> let bill = ("Bill", 2017, 3.2)
Prelude> let mary = ("Mary", 2018, 3.5)
Prelude> let students = [bill, mary]
Prelude> students
[("Bill",2017,3.2),("Mary",2018,3.5)]
```

These two tuples can be placed in the same list, because they contain the same types of values at the same positions within the tuple. The type of the resulting list seems complicated at first glance, but you can see that the GHCi has done its best to infer the component types from the values in the structure:

```haskell
Prelude> :type students
students :: (Fractional t1, Num t) => [(Char, t, t1)]
```

If you want to specify the types for the student record and the list of students in this example more narrowly, you can define some type synonyms, as follows:

```haskell
Prelude> type Student = (String, Int, Float)
Prelude> type StudentList = [Student]
Prelude> let bill = ("Bill", 2017, 3.2) :: Student
Prelude> :type bill
bill :: Student
```
Prelude> let mary = ("Mary", 2018, 3.5) :: Student
Prelude> let students = [bill, mary]
Prelude> :type students
students :: [Student]

The values contained within a tuple are usually accessed with pattern matching. For example, the following code extracts the name, class year, and GPA from the student record named *mary* and prints these in the GHCi:

Prelude> let (name, year, gpa) = mary
Prelude> putStrLn (name ++ "\n" ++ (show year) ++ "\n" ++ (show gpa))
Mary
2018
3.5

Pattern matching can also be used to access a student record within a list of students. For example, the next function expects the name of a student in the list and returns that student’s GPA:

\[
\text{findGPA :: String } \rightarrow \text{ StudentList } \rightarrow \text{ Float}
\]

\[
\text{findGPA name ((n, year, gpa):others)}
\]

| \( n == \text{name} = \text{gpa} \)
| \( \text{otherwise} = \text{findGPA name others} \)

Note that the pattern in this definition drills into the components of the tuple at the head of the list. Here is a run of the function with the list created earlier:

Prelude> findGPA "Bill" students
3.2
Prelude> findGPA "Mary" students
3.5

Although the tuple is a handy way to group together items of different types, it becomes unwieldy when used for complex structures. For one thing, you must remember the position of each type of value when constructing a pattern for matching a tuple. This quickly becomes a tedious and error-prone task. For another, quite random tuples may match your patterns, even though you’ve defined type names for them that supposedly make your code more readable. You will learn remedies for these shortcomings later in this book.

**Exercises**

1. Using the *Student* and *StudentList* types in this section, define a function
   \text{getStudentsInClassYear}. This function expects a class year and a *StudentList* as arguments and returns a *StudentList*. The result list includes the students belonging to the given class year.

2. Using the *Student* and *StudentList* types in this section, define a function
   \text{getHighestGPA}. This function expects a *StudentList* as an argument and returns a *StudentList* as well. The result list includes the student with the highest GPA, and may include multiple students if their GPAs are tied for the highest one.
List Comprehensions
You might call the list the workhorse of data structures in Haskell. When its items are characters, it becomes a string to be used in text processing. When its items are data structures of other types, such as tuples or lists of integers, it becomes the means of representing intricate objects such as matrices, association lists, and databases.

While it’s fun and easy to define recursive list processing functions for almost any task, there are some common, repetitive patterns that appear in such functions. One pattern is to transform a list of items into a list of the results of some operation on each item. The makeUppercase function, which applies the toUpper function to each character in a string, is one example.

Another pattern is to select some items that pass a test and reject others that don’t, before applying an operation and returning the results. For example, one might apply a curve to a set of grades after dropping all grades below a 60. In this section, you explore the list comprehension, a type of Haskell expression that simplifies the writing of both of these patterns.

The simplest form of the list comprehension takes each item from a list, uses it in an expression, and builds a list of the results. Its form is

\[
\{ <\text{expression}> \mid <\text{item}> \leftarrow <\text{source list}> \}
\]

In this form, \(<\text{item}>\) is a variable or a pattern, which is used in \(<\text{expression}>\) to produce a new value, for each item in \(<\text{source list}>\).

Here are some examples:

\[
\begin{align*}
\text{Prelude}\&\ >> [x ^ 2 \mid x \leftarrow [1..4]] \\
&= [1,4,9,16] \\
\text{Prelude}\&\ >> [\sqrt{x} \mid x \leftarrow [4,9,16]] \\
&= [2.0,3.0,4.0] \\
\text{Prelude}\&\ >> [\text{odd } x \mid x \leftarrow [1..4]] \\
&= [\text{True,True,False,False}] \\
\text{Prelude}\&\ >> [x + y \mid (x,y) \leftarrow [(1,2),(2,3),(3,4)]] \\
&= [3,5,7]
\end{align*}
\]

Note that the item to the left of the \(\leftarrow\) symbol can be a pattern which matches the type of item in the source list. Any components in the pattern can then be used in the target expression.

Armed with this simple form of list comprehension, you can simplify the makeUppercase function defined earlier, as follows:

```haskell
import Data.Char (toUpper)
makeUppercase :: String -> String
makeUppercase str = [toUpper x | x <- str]
```

You can also pull out the GPAs from the list of students defined earlier and compute the average GPA for this list, as follows:
averageGPA :: StudentList -> Float
averageGPA list =
    sum [gpa | (name, year, gpa) <- list] / length list

As you can see, the advantage of using the list comprehension is that you don’t have to
think of a recursive pattern every time you want to generate a list of results from a list of
values.

A more complex form of list comprehension adds one or more Boolean expressions,
which serve as a kind of guard, as follows:

[<expression>  | <item> <- <source list>,
    <Boolean expression-1,
    ...
    <Boolean expression-n>]

In this form of list comprehension, each item generated from the list is run through each
of the Boolean expressions. If the item passes all of these tests, it’s passed on to the
target expression, whose result is added to the list of results. Thus, the Boolean
expressions serve as a filter on the items to be processed. Here are a couple of examples
of its use:

Prelude> [x ^ 2 | x <- [1..10], odd x]
[1,9,25,49,81]
Prelude> [x ^ 2 | x <- [1..10], odd x, x > 4]
[25,49,81]

Alternatively, you can use a list comprehension simply to filter out unwanted items:

Prelude> [x | x <- [1..10], odd x]
[1,3,5,7,9]
Prelude> [x | x <- [1..10], x `mod` 3 == 0]
[3,6,9]

This part concludes with a discussion of a sort function that uses a quick sort. As the
name implies, the quick sort can run much faster than the insertion sort that was
developed earlier. The quick sort’s strategy for a nonempty list is to

1. Pick an item, called the pivot, and move the smaller items to its left and the larger
   items to its right. In the example that follows, you pick the first item as the pivot.
2. Sort the sublists of items to the left and right of the pivot.
3. Combine the results by appending three lists: the result of the first recursive sort,
   a list containing the pivot, and the result of the second recursive sort.

The second and third steps are trivial, but the first step seems like a daunting task. Here
is where the list comprehension comes to the rescue. To create each sublist for this step,
you take items from the entire list and retain those that pass a comparison test with the
pivot. Here is the code for the quickSort function:

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
Why does `quickSort` run faster than `insertionSort`?

Recall that the `insertionSort` function visits each item in the list and inserts it into the result of sorting the tail of the list after than item. If `insertionSort` visits each item in a list of length \( n \), it must run the `insert` function \( n \) times. The `insert` function searches for an item’s proper place in a list. In the worst case, where each item is inserted at the end of the list, the search requires \( m \) comparisons for a list of length \( m \). This is also true in the average case, because \( m/2 \) is approximately equal to \( m \), for very large \( m \). The first call of `insert` is passed an empty list, and each successive call gets a list whose length is one greater than that passed to the previous call. The final call of `insert` is passed a list where \( m = n \). Thus, the total number of comparisons that must be made in `insert` is \( (n^2 - n)/2 \) or approximately \( n^2 \), on the average.

The `quickSort` function takes a nonempty list of length \( n \) and runs two list comprehensions on it, for a total of \( 2n \) comparisons. For large \( n \), this is approximately equal to \( n \) comparisons. How many times must this be done? Ideally, consider the cases where the pivot item happens to be the median item in the list, all the way through the process. This means that each time `quickSort` is called, it will split the list into two equal parts and sort each half (minus the pivot each time). The total number of times you can split a list in two, before its length becomes 0, approximately equals \( \log_2 n \). Thus, in the ideal case, the `quickSort` function makes approximately \( n \log_2 n \) comparisons. For a list of 10,000 items in this case, `quickSort` will make about 132,877 comparisons, whereas `insertionSort` will make about 100,000,000 comparisons!

Unfortunately, the `quickSort` function also has a worst-case performance. Consider the case of a list that is already sorted. The pivot then always happens to be the smallest item, the one in the leftmost position. This means that the rightmost recursive call is run on a list whose length decreases by one each time the list is split, for a total of \( n \) subdivisions. Thus, in the worst case, the `quickSort` function performs approximately \( n^2 \) comparisons.

Although `quickSort` still performs better on the average than `insertionSort`, there are ways to avoid its worst-case behavior. For example, you can run another sort function, like `insertionSort`, within `quickSort`, when the length of the list reaches a minimum threshold.

**Exercises**

1. Redefine the `getStudentsInClassYear` function from the last set of exercises so that it uses a list comprehension instead of recursion.
2. Modify the `quickSort` function so that it runs an insertion sort on the list when the list’s length is less than or equal to 20 items.
3. Define a `sortStudentsByName` function that expects a `StudentList` as an argument and sorts the student tuples in the list by name. The function should return the sorted list, and use the quick sort algorithm discussed in this section.

```haskell
quicksort [y | y <= x] ++ [x] ++
quicksort [y | y > x]
```
Part III. Functions as Data: Higher-Order Functions and Function-Level Operators

In the functional model of computation, the basic unit of computation is of course the function. For simple tasks, a simple function call or application often suffices. For more complex tasks, you have seen how to

- compose expressions that consist of function calls
- break down a complex task into a set of subtasks and define functions to handle them

Both of these mechanisms provide ways to organize functions to solve complex problems.

Yet another way to organize code in functional programming is to exploit the idea that a function is also a special kind of data value. As such, a function can be passed as an argument to another function, or returned as its value, or placed in a data structure. Your code can then grab this passive datum and apply it to its arguments when the appropriate occasion arises. In this part of the book, you will learn how to treat functions as data, to further simplify solutions to complex problems.

Mapping

In Part II, you learned how to process data in a list by defining recursive functions or by using a list comprehension. One type of process transforms the items in a list into a list of results, by applying an operation to each item in the input list. For example, the next code segment returns a list of the square roots of the numbers in a list:

\[ \sqrt{x} \mid x <\text{list} \]

The list comprehension is syntactic sugar for a recursive function that does the same thing:

```haskell
roots :: Floating a => [a] -> [a]
roots [] = []
roots (x:xs) = sqrt x : roots xs
```

This pattern of computation is called a mapping. The only things that vary in its use are the list and the operation applied to each item. You can capture this pattern in a function that expects both the list and the operation as arguments, as follows:

```haskell
myMap :: (a -> b) -> [a] -> [b]
myMap _ [] = []
myMap f (x:xs) = f x : myMap f xs
```

This function essentially transforms a list of items of type \(a\) into a list of items of type \(b\). It uses a function \(f\) that takes an argument of type \(a\) and returns a value of type \(b\) (the type of the function is \((a -> b))\). Note that there are two references to the function \(f\) on the right side of the second clause of \(\text{myMap}\). The first reference applies \(f\) to its argument, \(x\). The second reference passes \(f\) as a datum to a recursive call of \(\text{myMap}\).
The `myMap` function is called a \textit{higher-order function}, because it takes another function as an argument. The use of higher-order functions can simplify your code; for example, compare the use of `myMap` with the list comprehension:

\[
\{\sqrt{x} \mid x \leftarrow \text{list}\}
\]

\texttt{myMap \sqrt \text{list}}

Haskell includes a built-in function named \texttt{map} for mapping functions onto lists of data. Here are some examples of its use:

\begin{verbatim}
Prelude> map \sqrt \{4,9,25\}
\{2.0,3.0,5.0\}
Prelude> map abs \{-5..5\}
\{5,4,3,2,1,0,1,2,3,4,5\}
Prelude> map \sqrt (map abs \{-25,-4,0,4,25\})
\{5.0,2.0,0.0,2.0,5.0\}
\end{verbatim}

List processing applications often feed the results of one transformation as inputs into another transformation. The \texttt{map} function provides a concise way to do this.

\textbf{Lambda Expressions}

You might think that a simple list comprehension has one advantage over the \texttt{map} function: the expression used to transform each value in the list comprehension need not be a function applied to a single argument, whereas the \texttt{map} function requires this:

\begin{verbatim}
Prelude> \{x \wedge 2 \mid x \leftarrow \{1..5\}\}
\{1,4,9,16,25\}
Prelude> map ? \{1..5\}
\end{verbatim}

One way around this problem is to define a helper function of one argument that performs the computation, as follows:

\begin{verbatim}
Prelude> let square x = x \wedge 2
Prelude> map square \{1..5\}
\{1,4,9,16,25\}
\end{verbatim}

This remedy is fair enough, but it forces you to define a one-off function each time you want to use \texttt{map} with a complex expression. Fortunately, Haskell provides a better way to handle this problem. It’s called a \textit{lambda expression}. The term “lambda” comes from Alonzo Church’s recursive lambda calculus (see the Introduction to this book). In a functional programming language, a lambda expression creates an anonymous function when it is evaluated. This function can then be passed to a higher-order function and applied later when needed.

Here is the form of a lambda expression in Haskell:

\begin{verbatim}
\langle \text{argument-1} \rangle \ldots \langle \text{argument-n} \rangle \rightarrow \langle \text{expression} \rangle
\end{verbatim}

The \langle symbol is as close to the Greek letter \textit{λ} as you can get with a Latin keyboard. The arguments can be variable names or patterns. Note that the \rightarrow symbol is used instead of the = symbol used in named function definitions.
Armed with lambda, you can map any expression you want onto the appropriate list:

```
Prelude> map (\x -> x ^ 2) [1..5]
[1,4,9,16,25]
map (\(x,y) -> x + y) [(1,2), (2,3), (3,4)]
[3,5,7]
```

Note that the entire lambda expression must be enclosed in parentheses when passed to `map`, so that `map` will know that it’s a single argument value. You can apply the same lambda expression to its arguments at the top level in the GHCi:

```
Prelude> (\(x,y) -> x + y) (1,2)
3
Prelude> :type (\(x,y) -> x + y)
\(x,y) -> x + y :: Num a => (a, a) -> a
```

**Exercises**

Define a function `listToString :: Show a => [a] -> String`. According to its type signature, this function expects a list of items that support the `show` function and returns a string. The `listToString` function should map the `show` function onto its argument list, and feed the result to the `join` function that you defined in an earlier exercise.

**Filtering**

Another feature of the list comprehension is its ability to select only certain items from a list for processing. These items must pass one or more tests; for example, you can retain only the even numbers in a list, as follows:

```
[x | x <- list, even x]
```

This expression is syntactic sugar for a recursive function that does the same thing:

```haskell
allEvens :: Integral a => [a] -> [a]
allEvens [] = []
allEvens (x:xs)
    | even x = x : allEvens xs
    | otherwise = allEvens xs
```

As with the mapping function, you can generalize this pattern to a higher-order function, which is known as a `filter`. This function takes a predicate (a Boolean function) as an argument and uses it to filter a list, as follows:

```haskell
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ [] = []
myFilter p (x:xs)
    | p x = x : myFilter p xs
    | otherwise = myFilter p xs
```

The filtering function essentially keeps an item in the list if it passes the test `p`, or removes it otherwise. As you might expect, Haskell already includes a built-in `filter` function that does the same thing. Here are some examples of its use:
Prelude> filter odd [1..10]
[1,3,5,7,9]
Prelude> filter (\x -> x `mod` 3 == 0) [1..27]
[3,6,9,12,15,18,21,24,27]
Prelude> filter (\x -> odd x && x `mod` 3 == 0) [1..27]
[3,9,15,21,27]
Prelude> map (\x -> x ^ 2) (filter odd [1..10])
[1,9,25,49,81]

As you can see, list processing can consist of a filtering followed by a mapping, or conversely.

**Exercises**

1. Redefine the `getStudentsInClassYear` function from an earlier exercise so that it uses a filter instead of a list comprehension.

2. Use a filter to define a function `removePunctuation`. This function expects two strings as arguments and returns a string. The second string argument includes the punctuation marks (the characters in the top row and right side of the keyboard) to be removed from the first string argument. *Hint*: the `filter` function should use a lambda expression that applies the `elem` function to the current character and the string of punctuation characters.

**Folding**

The `map` and `filter` functions are convenient if you prefer a more concise notation than the list comprehension. Another common pattern of list processing is the *reduction* of a list of items to a single item. For example, the built-in `sum` and `product` functions compute the sum and product of the numbers in a list:

```
Prelude> sum [1..10]
55
Prelude> product [1..10]
3628800
```

If these functions did not exist, you could not use a list comprehension or a map to define them. But you could provide the following recursive definitions, assuming that the list argument is nonempty:

```
mySum :: Num a => [a] -> a
mySum [x] = x
mySum (x:xs) = x + mySum xs

myProduct :: Num a => [a] -> a
myProduct [x] = x
myProduct (x:xs) = x * myProduct xs
```

Here is a trace of `mySum` when run with the list `[3,5,7]`:
mySum [3, 5, 7] ->
3 + mySum [5, 7] ->
5 + mySum [7] -> -- list has one item
<- 7 -- Returns 7
<- 12 -- Returns 5 + 7
<- 15 -- Returns 3 + 12

Note that mySum performs the additions of the results from right to left through the list, beginning with the last number and ending with the first one. Note also that the definitions of mySum and myProduct show the same pattern of code, with the exception of the function’s name and the operator used to perform the computation in the second clause. If you rewrite these clauses using prefix notation, you can see the possibility of generalizing this pattern to a higher-order function:

mySum :: Num a => [a] -> a
mySum [x] = x
mySum (x:xs) = (+) x (mySum xs)

myProduct :: Num a => [a] -> a
myProduct [x] = x
myProduct (x:xs) = (*) x (myProduct xs)

Folding from the Right

The higher-order function for reduction is also called a fold. The simplest version of this function expects a function of two arguments and a nonempty list as arguments. It repeatedly applies this function to pairs of values, one of which is the current head of the list and the other of which comes from a recursive call on the list’s tail. Here is the code for this function:

myFoldr1 :: (a -> a -> a) -> [a] -> a
myFoldr1 _ [x] = x
myFoldr1 f (x:xs) = f x (myFoldr1 f xs)

The term “foldr1” indicates that the function folds the values in the list, starting with the rightmost one, into one value. Haskell of course includes a built-in function named foldr1, as well as several other flavors of this function, named foldr, foldl, and foldl1. Because the folding pattern is very powerful and useful, you will explore the foldr function next and examine the others in a later section.

The foldr function extends the foldr1 function to include the case of the empty list. To do this, the foldr function requires a third argument – the value to return when the list is empty. Here is the code for a facsimile, named myFoldr:

myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr _ baseValue [] = baseValue
myFoldr f baseValue (x:xs) = f x (myFoldr f baseValue xs)

Note that this more general version allows the arguments to the argument function to be of different types (signified by a and b). Now you can define the functions mySum and myProduct in terms of myFoldr:
mySum :: Num a => [a] -> a
mySum list = myFoldr (+) 0 list

myProduct :: Num a => [a] -> a
myProduct list = myFoldr (*) 1 list

Here is a trace of myFoldr with the arguments (+), 0, and [3, 5, 7]:

myFoldr (+) 0 [3, 5, 7] -> 3 + myFoldr (+) 0 [5, 7] -> 5 + myFoldr (+) 0 [7] -> 7 + myFoldr (+) 0 [] -> -- List is empty <- 0 -- Returns 0 <- 7 -- Returns 7 <- 12 -- Returns 15 <- 15 -- Returns 15

Like any other argument, the function argument to myFoldr is like a switch you can set to specialize the function for particular applications.

Folding from the Left
The foldl function also folds a list of values into a single value, but performs the fold from left to right as it goes through the list. To accomplish this, the argument function is applied to the head of the list and the base value, and the result becomes the new base value for each recursive call. Here is the code for a facsimile, named myFoldl:

myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl _ baseValue [] = baseValue
myFoldl f baseValue (x:xs) = myFoldl f (f x baseValue) xs

Now you might think that a trace of a call of myFoldl to compute the sum of the list [3, 5, 7] would look like this:

myFoldl (+) 0 [3, 5, 7] -> -- base is 0
  myFoldl (+) 3 [5, 7] -> -- base is 3 + 0
    myFoldl (+) 8 [7] -> -- base is 5 + 3
      myFoldl (+) 15 [] -> -- base is 8 + 7
        <- 15 -- Returns 15
        <- 15 -- Returns 15

Unlike myFoldr, which performs the computations after each recursive call, myFoldl shows a tail-recursive process, where the computations appear to be performed when myFoldl’s arguments are evaluated. But before you conclude that folding from the left is less costly than folding from the right, you must realize that the appearances in this case are deceptive, and the above trace is quite inaccurate. Haskell functions do not evaluate their arguments until their values are needed within the function’s definition. This delay is called lazy evaluation. It’s very powerful and good for many things, but not in the case of functions that fold from the left. To see why, here is a more accurate portrayal of the trace:
myFoldl (+) 0 [3, 5, 7] -> -- Delay
myFoldl (+) (3 + 0) [5, 7] -> -- Delay
  myFoldl (+) (5 + (3 + 0)) [7] -> -- Delay
    myFoldl (+) (7 + (5 + (3 + 0))) [] -> -- Evaluate!
<- 15 -- Return 15
<- 15
<- 15

Thus, even though this function is tail-recursive, it incurs an increasingly expensive memory cost to track the expression that is passed as its second argument on the recursive calls. In the next section, you will learn more about lazy evaluation and its benefits as well as its costs.

Exercises
Modify the join function from an earlier exercise so that it uses a folding function rather than recursion.

Lazy Evaluation
In an earlier section, you learned that a function application runs an operation on its arguments and returns a value as a result. This behavior is true of function applications in any programming language; but the details of how this process works in Haskell are quite different than in most other languages.

In most programming languages, such as C and Python, the runtime system computes that values of all of a function’s arguments before applying the function’s code to these values. Consider, for example, the hypotenuse function introduced earlier:

```haskell
hypotenuse :: Floating a => a -> a -> a
hypotenuse a b = sqrt (a ^ 2 + b ^ 2)
```

In most other programming languages, when the sqrt function is applied to the sum of the squares of two values,

- the squares of the two values are computed first
- these two values are then summed
- finally this result is passed to the square root function for further processing.

Put another way, in most languages, the evaluation of a complex expression is eager, working from the inside out; the most deeply nested expressions that are arguments to functions in other expressions are evaluated first.

Lazy Days
By contrast, a Haskell function delays computing the values of its arguments until they are needed for further processing within the function’s code. In the hypotenuse example, these delays occur all the way down to the applications of the ^ operator. Thus, when the sqrt function is called, it receives as an argument not the value of the sum of the squares of the other two sides, but the entire expression that computes this value. When the code in sqrt needs this expression’s value, the + operator in this expression is applied to its two operand expressions, and a similar process of delayed evaluation occurs for them. The two exponentiation expressions are passed as arguments to the + operator, and
when its code needs their values, the ^ operator is applied to its operand expressions in a
similar manner. This process is called lazy evaluation, to indicate that the values of any
arguments or operands are not computed until they are needed, at any point in a
computation. Thus, in contrast to most other languages, Haskell expressions are
evaluated from the outside in.

For simple function applications like the hypotenuse example, you do not need to think
at all about lazy evaluation; the results of your code will be the same as those for function
applications written in other languages. However, for other function applications, like
those of the foldl function in the previous section, there can be significant costs that you
should beware of. The primary cost of lazy evaluation is that the runtime system must
reserve memory not for the values of argument expressions, but for the code of the
expressions themselves. These chunks of memory are called thunks. The memory cost of
creating thunks for some argument expressions, as you saw with foldl, can grow
directly with the size of a problem.

On the other hand, lazy evaluation has some benefits that are worth exploring a bit.

**Infinite Lists**

One benefit of lazy evaluation is its support for the use of infinite lists. As you probably
know, any program runs on hardware with a finite amount of memory to represent data.
Thus, the size of an integer or the length of a list is limited by the amount of memory
available. But it is possible to treat a list as an infinite sequence of items, if

- the runtime system is set up to move through this sequence and recycle memory
  as it goes
- function applications delay the evaluation of their arguments until their values are
  needed.

To explore this idea, consider the definitions of the built-in Haskell list processing
functions take and drop:

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

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

take returns a list of the first n items in a list, whereas drop returns a list of the items
after the first n items:

```
Prelude> take 3 [1..5]
[1,2,3]
Prelude> drop 3 [1..5]
[4,5]
```

Now consider what happens when you try to run take and drop on an infinite list, which
you can represent in Haskell with the notation [first integer ..]: 
The call of `take` halts with the correct result, whereas the call of `drop` does not halt, and you have to press control-c to end the computation.

Why does the `take` function halt? If you review its definition, the second clause of `take` matches the non-empty list argument to the pattern `x:xs`. But neither the head item `x` nor the tail of the list `xs` is evaluated until it is needed on the right side of the equation. Even then, only `x` is needed to add to the result of the recursive call of `take`, whereas `xs` is simply passed along as an expression to the recursive call. In other words, the `take` function simply strips an item off the head of an infinite list on each call, and stops when `n` of these have been taken and added to a result list.

The `drop` function, by contrast, must return the rest of the argument list after the first `n` items have been stripped off its head. In the code for `drop`, the second clause manages to strip these items away successfully before each recursive call. However, when `n` becomes 0, the first clause of `drop` must return the list argument with the remaining items. At that point, the runtime system attempts to complete the construction of the infinite list, which of course cannot be done on a computer with finite resources. The runtime system will eventually crash with a memory error, unless you halt the process by pressing control-c beforehand (you will observe the same phenomenon if you try creating an infinite list by entering `[1..]` directly at the GHCi prompt).

**Partial Function Application**

Another benefit of lazy evaluation is called *partial function application*. Consider the definition of the string function `makeUppercase` introduced earlier:

```haskell
let makeUppercase str = map toUpper str
```

Lazy evaluation allows you to shorten this definition to

```haskell
let makeUppercase = map toUpper
```

Recall that `map` is a higher-order function that expects another function and a list as arguments. When you omit the list argument in the second definition, the Haskell compiler creates code for a function that, when applied, will still expect a list as its argument. You can define a summation function with `foldr` in a similar manner:

```haskell
let mySum = foldr (+) 0
```

You can also do this for any binary operator, as in the following example:

```haskell
let incBy2 = (+) 2
```

This definition is functionally equivalent to

```haskell
let incBy2 x = (+) x 2
```
Currying
The technique of partial application of functions is made possible by a mechanism called *currying* (named after Haskell Curry, who was introduced earlier in this book). In the theory of functions that Curry developed, all functions have just one argument. Despite all appearances, this is true of functions in the Haskell programming language as well. When the compiler sees a definition of a function of two arguments, it creates code for a function of one argument (the first one) that returns another function of one argument (the second one). This process is generalized for a function of \( n \) arguments. You can infer this by examining the types of \( \text{map} \) and \( \text{map toUpper} \), as follows:

```
Prelude> import Data.Char
Prelude Data.Char> :type map
map :: (a -> b) -> [a] -> [b]
Prelude Data.Char> :type (map toUpper)
(map toUpper) :: [Char] -> [Char]
```

As you can see, each time you partially apply a function, you remove one type from the type signature of the resulting function. You are just making explicit what the Haskell compiler automatically does for you under the hood. Nevertheless, the use of partial function application can be a convenient way to make your code simpler and more concise.

Function Level Operators: Application and Composition
Thus far in this book, you have seen how to structure complex code in terms of cooperating functions, by using top-down or bottom-up design with stepwise refinement, and by using higher-order functions. Another technique to structure code, that exploits the concept of a function as a data value, is called *function composition*. In this case, you want to feed the value computed by one function as a data value, as shown in Figure 7.

**FIGURE 7**
Function composition

You can express the code to represent this process easily enough, as follows:

```
Prelude> let combination x = f (g x)
```

Note that the application of \( g \) to \( x \) requires parentheses. Without them, function \( f \) would be applied to two arguments, \( g \) and \( x \), which would produce an error. You can think of function application as left associative, meaning that each function is immediately applied to the item(s) to its right, so successive function applications in an expression move from left to right. This rule can lead to some unwieldy syntax, such as
Prelude> sqrt (abs (-3))
1.7320508075688772
Prelude> let bigCombination x = f (g (h (i x)))

The `bigCombination` function essentially passes data through four function applications, staring with function `i` and moving right to left. A convenient way to avoid this cumbersome notation is to use the function application operator `.`. Like the `^` operator, the `&` operator is right associative; consecutive applications of `&` are evaluated from right to left. Thus, you can use the `&` operator to make your code more readable:

Prelude> sqrt & abs & -3
1.7320508075688772
Prelude> let bigCombination x = f $ g $ h $ i $ x

You can simplify this type of code still further with Haskell’s function-level composition operator. This operator is just the period character (`.`). Here are some example uses:

Prelude> :type (.)
(·) :: (b -> c) -> (a -> b) -> a -> c
Prelude> let sqrtOfAbs = sqrt . abs
Prelude> sqrtOfAbs (-3)
1.7320508075688772
Prelude> let bigCombination = f . g . h . i

The composition operator allows you to define and glue together a set of functions that will work like an assembly line, through which data are passed for various transformations. As the type signature for the `.` operator shows, the type of argument to the first operand function must be the same type as the value returned by the second operand function. And you must remember that the data flow will be from right to left when the composition is applied. Thus, the function composition `f . g . h` will run the functions `h`, `g`, and `f` in that order when it is applied to its argument.
Part IV. Creating New Data Types

Haskell includes several basic data types, such as \texttt{Int}, \texttt{Float}, \texttt{Char}, and \texttt{Bool}, and two structured data types, the list and the tuple, for building data structures. You have built data structures composed of values of the basic types or the values of other data structures. You have also seen how to give new names to any data type to make your code more readable. For example, Haskell itself assigns the type name \texttt{String} to the data type \texttt{[Char]}, using the type synonym notation

\begin{verbatim}
type String = [Char]
\end{verbatim}

Finally, you have seen that each data type comes with a set of operations that apply to values of that type. The values and operations of a given type together comprise an \textit{abstract data type} (or ADT, for short). The term “abstract” means that the user of these values and operations does not know the details of how the values are represented and processed; that is the responsibility of the programmer who provides the data type and its operations.

Although the built-in basic types and structured types work well for many purposes, programmers often need to provide new data types, not just synonyms for existing ones. In this part of the book, you explore how this is done in Haskell with algebraic types and record types. In the process, you will learn how to define your own abstract data types.

\textbf{Algebraic Types}

An \textit{algebraic type} allows you to specify the values of a brand new type. An algebraic type gives the programmer a new type name as well as a means of constructing values of that type. The programmer then can provide further operations on the type for other programmers to use. There are three ways to create algebraic types, depending on the needs of your application.

\textbf{Enumerated Types}

The simplest form of an algebraic type is called an \textit{enumerated type}. The form of an enumerated type definition is

\begin{verbatim}
data <type name> = <value name-1> | … | <value name-n>
\end{verbatim}

The definition introduces a new type name, as well as one or more symbolic values of that type. The type name and the value names must be capitalized. For example, the built-in \texttt{Bool} type might be defined as

\begin{verbatim}
data Bool = False | True
\end{verbatim}

Now assume that you include the following enumerated types in the module file \texttt{Enumerations.hs}:

\begin{verbatim}
data Day = Monday | Tuesday | Wednesday | Thursday |
Friday | Saturday | Sunday

data PrimaryColor = Red | Green | Blue
\end{verbatim}

Then you can import one of the new types and try it out:
Prelude> import Enumerations (Day)
Prelude Enumerations> let day = Monday
Prelude Enumerations> let weekend = [Saturday, Sunday]
Prelude Enumerations> :type day
day :: Day
Prelude Enumerations> :type weekend
weekend :: [Day]
Prelude Enumerations> day
<interactive>:14:1:
    No instance for (Show Day) arising from a use of 'print'
    In a stmt of an interactive GHCi command: print it

Note that you can look up the type of your new values, as expected, but an error occurs when you try to display one of these values. This is because the GHCi does not know the string representation of your new symbolic values for printing. You can remedy this problem, and make your enumerated types more useful, by making them members of the type classes \texttt{Enum}, \texttt{Eq}, \texttt{Ord}, \texttt{Read}, and \texttt{Show}. To do this, you add the form

\texttt{deriving (<type class-1>, ... <type class-n>)}

after the type definition. Here is the modified definition of the \texttt{Day} type:

\texttt{data Day = Monday \mid Tuesday \mid Wednesday \mid Thursday \mid Friday \mid Saturday \mid Sunday \hspace{1cm} \text{deriving (Enum, Eq, Ord, Read, Show)}}

Deriving the \texttt{Show} class gives your new type a \texttt{show} function for free. This function returns the string representation of a value of your new type, and is run automatically when the GHCi needs to print this value. Deriving the \texttt{Eq} class gives your new type the \texttt{==} and \texttt{!=} operators for free. This operator allows you to compare your new values for equality. Deriving the \texttt{Ord} class allows you to use the comparison operators \texttt{<, >, \leq,} and \texttt{\geq} with your new values. Deriving the \texttt{Enum} class allows you to create lists of your new values using the \texttt{..} notation. You will learn about the \texttt{Read} type class and other type classes in the next part of this book. Here are some example uses of your new data type:

Prelude Enumerations> let day = Monday
Prelude Enumerations> day
Monday
Prelude Enumerations> show day
"Monday"
Prelude Enumerations> day == Tuesday
False
Prelude Enumerations> day < Tuesday
True
Prelude Enumerations> [Monday .. Friday]
[Monday,Tuesday,Wednesday,Thursday,Friday]

Note that the symbolic values of the new type are different in look and in type from their string representations.

You might be wondering by now why you would bother creating these new data types, when string values would appear to suffice instead. The primary reason for introducing
an enumerated algebraic type is type safety. Consider the following code, which represents a list of colors as strings, and then attempts to add a new color to it:

```
Prelude> let colors = ["Green","Blue"]
Prelude> "Apple" : colors
["Apple","Green","Blue"]
```

An apple might be red, but the apple itself is not its color. Restricting a type to a given set of symbolic values prevents the substitution of random strings for these values. The compiler will catch and flag any such attempts as errors.

You can put the enumeration of the days of the week to work by defining a new data type to represent the hours worked on each day for a given worker. This type is represented as a list of pairs. The first item of each pair is of type `Day` and the second item, the number of hours worked, is of type `Float`. Here is the definition:

```
type HoursWorked = [(Day, Float)]
```

To turn this type into a full-fledged abstract data type, you provide a set of operations to be used with it. These operations

- create a new value of this type
- look up the hours worked on a given day
- reset the hours worked on a given day to a new value
- compute the total hours worked for the week

Here is the code for the definitions of these functions:

```
-- Creates and returns a new HoursWorked value
newHours :: HoursWorked
newHours = map (\day -> (day, 0)) [Monday .. Sunday]

-- Returns the number of hours worked on a given day
hoursForDay :: Day -> HoursWorked -> Float
hoursForDay day ((d,h):xs)
  | day == d = h
  | otherwise = hoursForDay day xs

-- Resets the number of hours worked on a given day
setHoursForDay :: Day -> Float -> HoursWorked -> HoursWorked
setHoursForDay day newHours ((d,h):xs)
  | day == d = (d,newHours) : xs
  | otherwise = (d,h) : setHoursForDay day newHours xs

-- Returns the total hours worked on all days
totalHours :: HoursWorked -> Float
totalHours hr = foldr (\ (d,h) total -> h + total) 0.0 hr
```

Note the use of `map` and `foldr` in the definitions of `newHours` and `totalHours`, respectively. Although writing recursive definitions of functions is fun, you should avoid doing this when higher-order functions will work instead.
The function `setHoursForDay` changes the hours worked on a given day to a new value. It does so by creating and returning a new `HoursWorked` value from the values passed to it, rather than altering any of these values. This type of computation, which transforms a value into a new value, is consistent with the functional model.

Here are some sample uses of your new `HoursWorked` ADT:

```
HoursWorked> let hours = newHours
{[(Monday,0.0),(Tuesday,0.0),(Wednesday,0.0),(Thursday,0.0),
   (Friday,0.0),(Saturday,0.0),(Sunday,0.0)]}
HoursWorked> setHoursForDay Friday 7.5 hours
{[(Monday,0.0),(Tuesday,0.0),(Wednesday,0.0),(Thursday,0.0),
   (Friday,7.5),(Saturday,0.0),(Sunday,0.0)]}
HoursWorked> totalHours hours
0.0
*HoursWorked> totalHours (setHours Friday 7.5 hours)
7.5
*HoursWorked> hoursForDay Friday (setHours Friday 7.5 hours)
7.5
```

Product Types

The second form of algebraic data type is called a *product type*. Values of a product type consist of a type label and one or more component values. Here is its form:

```
data <type name> = <type label> <type-1> ... <type-n>
```

The type name and the type label are usually, though not always, the same. The type label also serves as a *constructor function* to create new values of a product type, using the following format:

```
<type label> <value-1> ... <value-n>
```

The types of the values must match the types in the product type definition.

For example, consider a new type to model an employee. An employee has a name, a pay rate, and a number of hours worked for each day of a week. Using the types `String`, `Float`, and `HoursWorked` for these components, you can define a new product type named `Employee` as follows:

```
Prelude> data Employee = Employee String Float HoursWorked
```

Now you can create a new value of this type for a given employee:

```
Prelude> let kenInfo = Employee "Ken" 15.00 newHours
```

Note the use of the `newHours` function to create a value of the `HoursWorked` type, which you defined earlier.

You can access the value of each component of a product type by pattern matching:

```
Prelude> let (Employee name payRate hours) = kenInfo
Prelude> name
```

"Ken"

Prelude> payRate
15.0
Prelude> hours
[(Monday,0.0),(Tuesday,0.0),(Wednesday,0.0),(Thursday,0.0),
(Friday,0.0),(Saturday,0.0),(Sunday,0.0)]

Note that the pattern (Employee name payRate hours) includes the type label as well as variables for the component values.

Even better, you can turn the Employee type into an abstract data type, like you did for the HoursWorked type. To do this, you define a set of operations for the Employee type. These operations go in the Employee module. This module imports the type and function names from the HoursWorked module, being careful to qualify the setHoursForDay and totalHours functions to avoid name conflicts between the two modules. Here is the code for the new module:

module Employee where

import HoursWorked (Day, HoursWorked, newHours)
import qualified HoursWorked (setHoursForDay, totalHours)

-- The data type of all employees, which includes a name, payrate
-- and hours worked
data Employee = Employee String Float HoursWorked
    deriving Show

-- Creates and returns a new Employee value
newEmployee :: String -> Float -> Employee
newEmployee name payRate = Employee n
    ame payRate newHours

-- Returns an employee's name
employeeName :: Employee -> String
employeeName (Employee name _ _) = name

-- Returns an employee's payrate
employeePayRate :: Employee -> Float
employeePayRate (Employee _ payRate _) = payRate

-- Resets the employee's payrate to a new rate
setEmployeePayRate :: Float -> Employee -> Employee
setEmployeePayRate newPayRate (Employee name _ hours) =
    Employee name newPayRate hours

-- Resets the employee's hours on a given day to a new value
setHoursForDay :: Day -> Float -> Employee -> Employee
setHoursForDay day newHours (Employee name payRate hours) =
    Employee name payRate (HoursWorked.setHoursForDay day
    newhours hours)

-- Returns the employee's total hours worked on all days
totalHours :: Employee -> Float
totalHours (Employee _ _ hours) = HoursWorked.totalHours hours
-- Returns the employee’s total pay on all days
totalPay :: Employee -> Float
totalPay employee =
  employeePayRate employee * totalHours employee

Several points about this code call for comment:

- The `setHoursForDay` and `totalHours` functions are imported as qualified from the `HoursWorked` module, to allow their use with the type qualifier `HoursWorked`. This technique avoids name conflicts between functions defined in the `HoursWorked` module and functions defined in the `Employee` module.
- The `Employee` type derives the `Show` type class, which allows you to print `Employee` values in the GHCi during testing. This derivation works as long as each of the component types in `Employee` also derive `Show`.
- The `newEmployee` function creates a new value of type `Employee` with the given name and pay rate as arguments, and a default value for the hours worked. You should always provide a function to construct a new value in your ADTs.
- Several functions use pattern matching on the `Employee` argument to extract the relevant component value, while ignoring the other components.
- The `Employee` function `setEmployeePayRate` runs a pattern match to obtain the values of the other components, and combines these with the new pay rate in creating a new value of type `Employee`.
- The `Employee` functions `setHoursForDay` and `totalHours` call functions with the same name on the hours worked component, using the qualified names `HoursWorked.setHoursForDay` and `HoursWorked.totalHours`.
- The `totalPay` function simply calls other `Employee` functions to compute its result, rather than drilling into its argument with a pattern.

**Union Types**

Occasionally, you will need an algebraic type that mixes the product and enumeration types. This is called a union type, to signify that either one or the other component will be present in a given value, but not both. A good example of a union type is Haskell’s built-in `Maybe` type. Here is the definition of `Maybe`:

```haskell
data Maybe a = Nothing | Just a
  deriving (Eq, Show)
```

Note the type variable `a`, which appears after the type name `Maybe` and after the type label/constructor `Just`. This definition says that a value of type `Maybe` can be either `Nothing` or a value of any type labeled by `Just`. Thus, a value of type `Maybe` expresses an optional value, where one option is actual data and the other option is a symbol, `Nothing`, indicating no data at all. Here are some examples:

```haskell
Prelude> Nothing
Nothing
Prelude> :type Nothing
Nothing :: Maybe a
Prelude> Just "Ken"
Just "Ken"
```
As with values of product types, the data in a value of type `Maybe` can be extracted during pattern matching.

Optional values are useful in search functions. For example, the module `Data.List` includes a `lookup` function that expects a string and an association list as arguments. An association list is a list of pairs of strings. The first string in each pair is called a `key`, and the second string is called a `value`. The `lookup` function is supposed to return the value that is associated with a given key. The key argument is compared to the first item in each pair in the list, until a match occurs or there are no more pairs to examine. If the latter is the case, the value `Nothing` is returned. Otherwise, the value `Just x` is returned, where `x` is the second string in the matching pair. Thus, function `lookup` returns a value of type `Maybe`. Here is an example:

```
Prelude> let aList = ["Name","Ken"],("Age","64")]
Prelude> lookup "Name" aList 
Just "Ken"
Prelude> lookup "Height" aList 
Nothing
```

Although the `lookup` function works as expected, there is a serious problem with this representation of association lists. An association list should contain unique keys; however, this representation allows any random list of pairs to serve as an association list, including lists with duplicated pairs. The remedy is to turn the association list into an abstract data type, by

- Defining the association list as a new algebraic type
- Including operations to create a new association list, add keys, replace values at given keys, and remove keys

By doing this, you insure that your clients will use an association list correctly.

Here are the definitions of the new `AssociationList` type and the function `lookup`:

```
data AssociationList = AssociationList [(String,String)]

lookup:: String -> AssociationList -> Maybe String
lookup key (AssociationList list) = helper list where
  helper :: [(String,String)]
  helper [] = Nothing
  helper ((k,v):pairs)
    | key == k = Just v
    | otherwise = lookup key pairs
```

Note that the `lookup` function unwraps the list within the `AssociationList` value and passes it to a helper function for the search.
To guarantee that an association list contains only unique keys, you provide functions to create a new list and to add a new key/value pair to it. The latter function adds the pair only if the key is not already present in the list. If the key is already exists, the function instead replaces the value at the matching key/value pair with a new value. Here is the code for these two new functions:

```haskell
-- Creates a new association list
newAssociationList :: AssociationList
newAssociationList = AssociationList []

-- Inserts a new key/value pair if the key is not in the list
-- Otherwise, replaces the value at the given key with the new value
addKey :: String -> String -> AssociationList -> AssociationList
addKey key newValue (AssociationList list) =
  AssociationList (helper list)
  where
    helper :: [(String, String)] -> [(String, String)]
    -- Add new pair
    helper [] = [(key, newValue)]
    -- Replace value in existing pair
    helper ((k,v):pairs)
      | key == k = (k, newValue) : pairs
      | otherwise = (k,v) : helper pairs
```

As in the definition of the `lookup` function, the `addKey` function first unwraps the list and then passes it to a helper function to insert or replace the target pair. The resulting list is then wrapped in a new `AssociationList` value for return to the caller. The `removeKey` function has a similar structure and is left as an exercise for you.

**Exercises**

1. Define a new type named `Book` to represent books. The components of a book are a title, an author, and a Boolean flag to indicate whether or not a book has been checked out.
2. Define a set of functions to manipulate the `Book` type of the previous exercise. The function `newBook` returns a book with the given author and title (it’s not checked out by default). The functions `author`, `title`, and `isCheckedOut` access and return the values of a book’s components. The function `checkOutOrReturn` resets the `isCheckedOut` component to its logical negation.
3. Use the `Data.List.lookup` function in the definition of the `lookup` function for the `AssociationList` type.
4. Define the `removeKey` function for the `AssociationList` type.
5. Define two functions, `keys` and `values`, for the `AssociationList` type. The `keys` function returns a list of keys, and the `values` function returns a list of values.
**Record Types**

Although algebraic data types certainly improve on tuples for modeling data structures, they can become unwieldy as the number of components in a structured type increases. Either you must remember the position of each component when accessing even a single one of them via pattern matching, or you must write a slew of functions to access or modify each component in an abstract data type.

It would be handy to create a new algebraic data type whose components have names as well as data types, so you could access or modify them by name rather than position. Fortunately, Haskell includes a mechanism for defining a new algebraic type as a *record type*. A record type provides a structured value containing named components, and also a free set of functions to access these components.

For example, consider the `Employee` type defined in the last section:

```haskell
data Employee = Employee String Float HoursWorked deriving Show
```

This type has three components, showing just their data types in the definition. Here is a rendition of the same algebraic type as a record type:

```haskell
data Employee = Employee
  { employeeName :: String,
  employeePayRate :: Float,
  employeeHours :: HoursWorked
  } deriving Show
```

This notation shows component names as well as their data types. A component name is like a variable within the record structure that refers to the value of a given component.

When you define this type, you also get three accessor functions with the same names as the corresponding components. The behavior and types of the `employeeName` and `employeePayRate` functions are exactly the same as those of the functions you defined in the previous version of the `Employee` ADT. Moreover, you get a constructor notation that uses position-independent, named components as well. Thus, you can redefine the `newEmployee` function as follows:

```haskell
newEmployee :: String -> Float -> Employee
newEmployee name payRate = Employee
  { employeeName = name,
    employeePayRate = payRate,
    employeeHours = newHours
  }
```

Note that the definitions of the component names do not have to appear in this particular order; any order will do.

Finally, the record notation allows you to simplify the code for modifying the individual components of a structured value. For example, the function that modifies an employee’s pay rate,
setEmployeePayRate :: Float -> Employee -> Employee
setEmployeePayRate newPayRate (Employee name _ hours) =
  Employee name newPayRate hours

can be redefined using the record notation as

setEmployeePayRate :: Float -> Employee -> Employee
setEmployeePayRate newPayRate employee =
  employee {employeePayRate = newPayRate}

The code for the new version of setEmployeePayRate digs into the existing record and selectively resets an individual component to a new value. The values of the other record components remain the same.

The use of records with named components can be really helpful to the writer and the reader of large structured types. Of course, pattern matching works just as before, when you have to resort to it in function definitions that process these values.

**Exercises**
Redefine the Book type and its associated functions from the last set of exercises, so that it uses the record notation.
Part V. Generic Data Types and Type Classes

In Part I and Part III of this book, you learned how to organize computational processes in terms of cooperating functions. In Part II and Part IV, you learned that organizing the data to be processed is equally important and useful. Now it’s time explore two concepts that have already played a role in much of the code you have seen: generic data types and type classes. You will see how both concepts help you to conceive and organize functions and data types that are effective and widely applicable.

Generic Types

A generic data type is a data type whose definition includes one or more type variables. These variables serve as parameters that can be replaced with actual data types when the generic type is used, in much the same way as function parameters can be replaced with actual argument values when the function is called.

You first encountered a generic type when you ran the GHCi type command with list operations such as length and !!:

```
Prelude> :type length
[a] -> Int
Prelude> :type (!!)
Int -> [a] -> a
```

The type variable a in the list type [a] indicates that a list is a generic type, because its item type can be any type whatsoever. The definitions of the length and !! operations do not need to know anything about a list’s item type, because they are concerned only with the list’s structure. Therefore, there need only be one definition of each of these operations for lists of any item type. For this reason, the length and !! operations are called polymorphic functions (the term comes from the Greek polymorphē, meaning many bodies).

You have also seen how to specialize the generic list type by substituting an actual data type for the type variable. For example, Haskell creates a type synonym for strings as follows:

```
type String = [Char]
```

The length, !!, and other polymorphic list functions still work on this special type of list, but Haskell also provides you with several string-specific functions, such as the words function:

```
Prelude Data.String> :type words
words :: String -> [String]
```

In this code, String is a synonym for [Char], and [String] is a synonym for [[Char]]. The map function is another example of a polymorphic function. This function expects another function and a list as arguments and returns a list. The input list is of one item type, and the output list can be of a different item type than the input list. The function
argument’s argument type must therefore be the same as the input list’s item type, and the function argument’s result type must be the same as the output list’s item type:

```haskell
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
```

Another generic type is the tuple. Like lists, tuples can hold items of any type; but unlike lists, there can be different types of items in the same tuple. Consider the built-in `zip` function, which takes two lists as arguments, creates pairs of items from these lists, and places them in a list of pairs:

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

This function’s signature includes two type variables, `a` and `b`, because the item types of the two lists need not be the same. And note that the function’s code need not know anything about types `a` and `b`, so the `zip` function is polymorphic as well.

**Exercises**

1. Define a function `myUnzip`, which is the inverse of the `zip` function. Your new function expects a list of pairs as an argument and returns a pair of lists. Be sure to make this function polymorphic, by using type variables in its signature.

2. The `AssociationList` type discussed in Part III restricts the types of keys and values to type `String`. Someone complains that the type of values in such a list should be any type of thing at all, including functions. For example, one should be able to store `Student` records in a list keyed by the student’s names or ID numbers, or functions in a list keyed by the functions’ names. Redefine the `AssociationList` type, including its functions, to accommodate this request, and use the type with a couple of examples.

**Type Classes**

Lists and tuples are generic data types, because their structures are independent of the types of items contained in them. Lists can be specialized to lists of characters, which are just like strings, or lists of pairs of strings, which are just like association lists. You can then define or use operations specialized for just these types.

However, you often want to use or define operations on a type of data that is not completely generic, like a list, but is not totally specific, like a character. That is, there might be a set of operations, like arithmetic, that work with numbers of different types, but not with values of any data types whatsoever. In other words, these operations could work with a family of data types, but not with any other data types. In Haskell, such a family of data types is called a type class. Haskell allows you to use and define type classes to restrict the range of allowable data types that some operations can manipulate. The data types that belong to a type class are called its members or instances.

You have seen and used many type classes already. Consider the type signatures of the `+`, `==`, `<`, and `Data.List.sort` operations:
The + operator is defined not just for integers, or for floating-point numbers, or for rational numbers, but for all of these types in so far as they belong to the Num type class. The == operator works with values of any type that belongs to the Eq type class. The < operator’s operands must be of a type belonging to the Ord type class, while the sort function works only with a list of such items. Some of the important built-in type classes and their instance types are listed in Table 8.

**TABLE 8**

Some Standard Type Classes and Their Instances

<table>
<thead>
<tr>
<th>Type Class</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum</td>
<td>Bool, Char, Double, Float, Int, Rational</td>
</tr>
<tr>
<td>Eq</td>
<td>Bool, Char, Double, Float, Int, Rational, String</td>
</tr>
<tr>
<td>Num</td>
<td>Double, Float, Int, Rational</td>
</tr>
<tr>
<td>Ord</td>
<td>Bool, Char, Double, Float, Int, Rational, String</td>
</tr>
<tr>
<td>Read</td>
<td>Most types, except for functions</td>
</tr>
<tr>
<td>Show</td>
<td>Most types, except for functions</td>
</tr>
</tbody>
</table>

The presence of a type class in a function signature ensures that the function will apply only certain operations to its arguments, even though the arguments’ data types appear to be generic. For example, the list passed to the sort function cannot contain just any old items, as it could for the length function. These items must be of an ordinal data type, as signified by the Ord type class, because the sort function will be comparing them to each other in its code. If the list’s item type is not a built-in type, such as Int or String, which already belongs to the Ord class, it is the responsibility of the item type’s programmer to guarantee that it does so. Any type that belongs to the Ord class must include definitions of the comparison operations required by that class.

Your data type can become a member of a type class in two ways: either by deriving the type class or by instantiating the type class

You have seen several examples of type class derivation. For example, the type

```haskell
data Day = Monday | Tuesday | Wednesday | Thursday |
        | Friday  | Saturday | Sunday  |
        deriving (Enum, Eq, Ord, Read, Show)
```

causes the algebraic type Day to join five different type classes. The benefit of type class derivation is that the compiler generates the required operations for each type class of your new type for free. You can usually get away with type class derivation for simple
enumerations like this one, or for structured types like lists and tuples, whose item types already belong to the same type classes.

You must resort to type class instantiation when these circumstances do not hold, or when you need to provide a custom definition of one or more operations required by a type class. A couple of simple examples will show how this is done.

The simplest type class to explore first is the `Show` class. This class requires you to include a function, named `show`, whose type signature is

```haskell
show :: Show a => a -> String
```

This function converts a value of your data type `a` to its string representation. Now, suppose your `Day` type does not derive the `Show` class. Then, in order to be able to do simple things like print values of type `Day` in the GHCi, you will need to make the `Day` type an instance of the `Show` class, as follows:

```haskell
instance Show Day where
    show Monday = "Monday"
    show Tuesday = "Tuesday"
    show Wednesday = "Wednesday"
    show Thursday = "Thursday"
    show Friday = "Friday"
    show Saturday = "Saturday"
    show Sunday = "Sunday"
```

Writing all these clauses is rather tedious, and is reason enough to use type class derivation! But suppose you want to show an employee’s information for a value of the type `Employee` developed earlier. Derivation might work, but might also give you a messy string with embedded list and tuple delimiters, among other things. Instead, you can instantiate the `Show` class with a `show` function that converts an employee structure into a nicely formatted string, as follows:

```haskell
instance Show Employee where
    show (Employee name payRate hours) = 
        "Name: " ++ name ++ "\n" ++
        "Pay rate: " ++ show payRate ++ "\n" ++
        "Hours worked:\n" ++ show hours
```

This code will build a string with each of the components of an employee structure, down to the hours worked per day, on separate lines, suitable for display in the GHCi or terminal. Note the extra calls of `show` with the pay rate and the hours, which are not already strings, but a floating-point number and another structure, respectively. This code assumes, of course, that the `HoursWorked` data type also instantiates the `Show` class with its own custom `show` function.

Another type class that is easy to instantiate is the `Eq` class. This class guarantees that its family of data types supports the `==` and `=/=` operations. Although many built-in data types are members of `Eq`, not all of them are, and some of your new data types might not be derivable. It is important for a data type to belong to this type class, so you can put values of this type into lists and search them with functions like `elem`:
Prelude> elem 5 [1..10]
True
Prelude> :type elem
elem :: Eq a => a -> [a] -> Bool

This function requires that the target item and the items in the list both be of type \texttt{a}, where this type supports the \texttt{==} operation.

Now consider updating the \texttt{Employee} type so you can search lists of employees for a given employee. You will need to instantiate the \texttt{Eq} type class and include an \texttt{==} operation. But when are two employee structures equal? You could compare each pair of components in the two employees, all the way down, but it’s simpler to assume that they’re equal if their names match (realistic databases would use an ID number). Here is the code for the instantiation:

\begin{verbatim}
instance Eq Employee where
  emp1 == emp2 = employeeName emp1 == employeeName emp2
\end{verbatim}

You don’t need to include a definition of the \texttt{/=} operation, because a default definition is provided in the \texttt{Eq} class. Here is the code for this class:

\begin{verbatim}
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
\end{verbatim}

Note that the \texttt{Eq} class provides defaults for both operations, so that the programmer who instantiates it need only supply one of them to get the other one for free.

One other important type class is the \texttt{Read} class. This class specifies a function named \texttt{read}, which is the inverse of the \texttt{show} function. Here are some examples of its use:

\begin{verbatim}
Prelude> :type read
read :: Read a => String -> a
Prelude> read "22" :: Int
22
Prelude> read "3.14" :: Float
3.14
Prelude> read "True" :: Bool
True
\end{verbatim}

The \texttt{read} function takes a string as an argument and wrangles it back into the data value that the string represents. To avoid ambiguity, the \texttt{read} function must also be supplied with the data type of the target value, so it can locate the appropriate instance of the conversion function. The \texttt{read} function is most easily used with and instantiated for simple types such as enumerations, but some structured types can include nested \texttt{reads} to obtain new structures from text files.

Although most of this part of the book has been concerned with the mechanics of using generic data types and type classes, you should not lose sight of their purpose. When you
define new functions and data types, you should aim for the mean between two extremes, regarding data types. That is, you should aim to make the data types of a function’s arguments and result values as general as possible, constraining them, if need be, only by the nature of the processing. A good example of this rule of thumb is the decreasing generality, or increasing specificity, of the functions `length`, `elem`, `Data.List.sort`, and `Dara.String.words`. What types of items must there be in a list for these functions to work correctly? And how can you enforce these constraints while allowing the widest range of data items in each case?

**Exercises**
The `AssociationList` type updated in the last set of exercises restricts its keys to type `String`. Someone complains that the type of key in such a list should be any instance of the type class `Eq`. For example, one should be able to create a list of integers keyed by strings, or a list of strings keyed by integers. Redefine the `AssociationList` type, including its functions, to accommodate this request, and use the type with a couple of instances.
Part VI. Interacting with the World: Terminal I/O and Files

Thus far in this book, you have learned how to structure code in terms of cooperating functions and data types, using either existing functions and data types or defining new ones of your own. You have also become familiar with the functional model of computation, wherein data are transformed as they pass through a series of function applications.

Throughout this process, you have run code in the GHCi, an interactive environment that allows you to supply input values to your functions and view their results. This process is different from running your code in standalone applications, which you launch from your computer’s operating system by tapping an icon, double-clicking on it with a mouse, or keying in a command at a terminal prompt. Such applications are also called programs, to distinguish them from expressions or function applications. A program requires additional features to interact with the external world, such as operations that take inputs from the keyboard and display outputs to a monitor, or operations that access data stored in your file system. A realistic program accepts input data, processes those data (perhaps using just the kinds of cooperating functions and data types you have seen thus far), and outputs the results.

In this section, you will learn about some of these features, particularly the input and output operations, first in the usual context of the GHCi. You will then learn how to incorporate them, together with the kind of functional code you have already seen, in complete standalone applications or programs.

Interactive Input and Output

When the GHCi needs to display the value of an expression, it implicitly calls the function putStrLn to do so. You have seen that you can also call this function explicitly in the GHCi, to force an output of a string. Here are some sample uses of this function and its sister function, putStr, which omits a newline at the end of its output:

```haskell
Prelude> putStrLn "Two lines\nof text."
Two lines
of text.
Prelude> putStr "No newline!"
No newline!
Prelude> :type putStrLn
putStrLn :: String -> IO ()
Prelude> :type putStr
putStr :: String -> IO ()
```

The type signatures of these functions show a String argument, as expected, but the return type, IO (), seems a bit odd. IO is the name of a type class that specifies various functions for input and output, and the () symbols here designate the empty type. The empty type is used as the instance of the IO type class for putStrLn and putStr, because neither function returns a value. This too is a bit odd; every function you’ve seen thus far returns a value of some kind. But output functions are run only for their side effects, namely, to send output to the external world (in this case, to the GHCi window). No caller is waiting to receive a value from such a function.
The input function `getLine` is the inverse of `putStrLn`. The `getLine` function pauses computation to wait for the user to enter a line of text. When the user presses the enter or return key, the function builds a string from the text and returns to the caller. Here is an example use of `getLine` (the user’s inputs are in italics):

```
Prelude> getLine
Ken Lambert
"Ken Lambert"
Prelude> getLine
Haskell Curry
"Haskell Curry"
Prelude> :type getLine
getLine :: IO String
```

Note that after the user enters his name on the second line of code, the GHCi displays this text as a string on the third line. Although the return type of `getLine` appears to be `String`, this type is also an instance of the `IO` type class.

Another thing that is clear from this example is that `getLine` is not a pure function: different calls can return different values, even though the arguments (there are none in this case) are the same. For this reason, functions like `getLine` are also called impure functions.

The best way to view a function that returns a type `IO a` is that the function performs some action in the context of a sequence of such actions. For example, a sequence of outputs and inputs might prompt the user for some input data. After the inputs have been received, they are transformed into results, perhaps by pure functions of the sort that you have already seen in this book. The results are then output by another sequence of actions. The complete sequence of such actions is a program. Programs in this sense fit within an imperative model of computation, to indicate that the computation proceeds through a sequence of commands or statements rather than by composing a series of function applications.

Haskell provides a fairly clean way of layering the imperative model on top of the functional model. The main point is to insulate the pure functions within the functional model from the impure functions that produce side effects in the imperative model. For now, you must keep in mind the following of rules of thumb when mixing together pure functions and impure functions:

1. A function of type `IO a` can only be called within the definition of another function of type `IO a` (although `a` does not have to be the same type in both cases). This means that you cannot call a function of type `IO a` within the definition of a pure function.
2. You can still call pure functions within the definitions of functions of type `IO a`.

You will apply these rules of thumb shortly.

If you want to obtain a sequence of input values, you must save them in variables for further processing. However, you cannot use Haskell’s `let` expression for this purpose. Because the `let` expression is a pure function application, you cannot include a call of
getLine within it. Instead, you use the following form to name the string obtained by getLine:

<variable name> <- getLine

Here is a brief example:

Prelude> firstName <- getLine
Ken
Prelude> lastName <- getLine
Lambert
Prelude> let fullName = firstName ++ " " ++ lastName
Prelude> putStrLn ("Your full name is " ++ fullName)
Your full name is Ken Lambert

The <- operator extracts a value of type String from the action performed by the getLine function. The GHCi implicitly uses <- to obtain the string result when getLine is called at the top level. Note that you can still use the pure code of the let expression to define a new variable within this sequence of actions.

Now that you’ve observed the behavior and learned the rules of thumb for using the simple I/O functions in the GHCi, you can turn the last example into a standalone program. To do so, you define a function named main within a module named Main (the file name in this case can be different from the module name). To the sequence of actions within this function, you add some outputs of prompts, to make the program’s behavior more interactive for the user. Here is the code for this module:

{-
 File: SimpleIO.hs
 -}

module Main (main) where

main :: IO ()
main = do
  putStrLn "Enter your first name: "
  firstName <- getLine
  putStrLn "Enter your last name: "
  lastName <- getLine
  let fullName = firstName ++ " " ++ lastName
  putStrLn ("Your full name is " ++ fullName)

There are just two things to note about this code:

1. The main function’s type must be IO (), because you are running other IO functions within its definition.
2. The sequence of actions is nested within a do block. This form allows the compiler to view expressions as a sequence of actions rather than pure function applications (although one of them, the let expression, remains pure).

Now you can test this program in the GHCi before you compile it as a standalone application:
Simple interactive programs all have this structure, and you can use this example as a template for them. You can include control expressions, such as the **if/then/else** and **case** expressions, as long as you enclose the actions for the alternatives within their own **do** blocks. For example, this program prompts the user for a password and prints an error message if she gets it wrong:

```haskell
{-
File: Password.hs
-}

module Main (main) where

main :: IO ()
main = do
  putStrLn "Enter your password (hint – it's 'Ken'): ".
  password <- getLine
  if password == "Ken" then do
    putStrLn "You're in!"
  else do
    putStrLn "Error: incorrect password!"
```

Here is a **GHCi** session that exercises both alternatives:

```haskell
Prelude> :load Password
[1 of 1] Compiling Main ( Password.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Enter your password (hint – it's 'Ken'): Ken
You're in!
*Main> main
Enter your password (hint – it's 'Ken'): Lucy
Error: incorrect password!
*Main>
```

You will learn how to compile and run a standalone application in the next section, which will also reveal a problem with the **getLine** function when it leaves the friendly confines of the **GHCi**. This in turn will motivate you to define some custom input functions.

**Exercises**

Modify the password program so that it repeats the prompts until the user enters the correct password.
Defining Custom Input Functions

As shown earlier in the book, you compile a standalone Haskell program by running the `ghc` command at a terminal prompt. Here is a compilation and execution of the `SimpleIO` program:

```
lambertk$ ghc --make SimpleIO.hs
[1 of 1] Compiling Main             ( SimpleIO.hs, SimpleIO.o )
Linking SimpleIO ... 
lambertk$ ./SimpleIO
Ken Lambert
Enter your first name: Enter your last name: Your full name is Ken Lambert
lambertk$
```

Note that the interaction with the user is not the same as it was when you ran the program in the GHCi. The outputs of the prompts are delayed until after the poor user guessed what to do and has entered the inputs. Not good!

The reason for this is that the terminal, unlike the GHCi, does not display the string that is output by either call of the `putStr` function, until a newline is output as well. But the text of these strings is gathered up in the terminal’s output buffer, to await the completion of the input operations and the detection of a newline in the buffer after the call of `putStrLn` at the end of the program.

You can remedy this problem by using `putStrLn` to output each prompt as well, but that will cause the inputs to be echoed on the lines below the prompts, rather than to their right. What you need is a way of flushing the text from the output buffer after each call of `putStr`, so that the prompt can be displayed on the same line as its input.

Instead of making every interactive program deal with this problem, you should package the solution in a customized input function. This function, named `getString`, can also display the prompt for the input. Thus, the function expects a string (the prompt) as its argument and returns the input string. But since the function uses other functions that return an `IO` type, it too must return an `IO` type. Here is the code, followed by some comments:

```
getString :: String -> IO String
getString prompt = do
  putStr prompt
  hFlush stdout
  inputString <- getLine
  return inputString
```

There are two new things to note about this function:

1. The function calls the `hFlush` function on the variable for the standard output stream, `stdout`, after the output of the prompt. `hFlush` flushes any text remaining in the output buffer, so this text will be displayed before the input function pauses the program for input.
2. The function appears to return the input string to its caller with a return expression at the end of the code. The purpose of return is the inverse of the <- operator that you saw earlier. return makes a value within an action available to the sequence of actions in which that action occurs, while <- extracts a value from a given action for further use in a sequence of actions.

Armed with getString, as defined in a module named TerminalIO, you can now correct the user interaction problems in the earlier programs. Here is the code for the modified SimpleIO program:

{-
 File: SimpleIO.hs
-}

module Main (main) where

import TerminalIO (getString)

main :: IO ()
main = do
  firstName <- getString "Enter your first name: 
  lastName <- getString "Enter your last name: 
  let fullName = firstName ++ " " ++ lastName
  putStrLn ("Your full name is " ++ fullName)

Note that a let expression appears as an action in the sequence of actions in the do block. You still must use a let expression when you want to give names to values computed by functions like ++ with no side effects. However, when let appears in a sequence of actions, you cannot not use the corresponding in clause.

Now consider programs that must input numeric values. These values, like any others, must first be input as strings, and then converted to values of the relevant numeric data types, such as Int or Float. You earlier saw how to use the read function to perform such conversions from the strings that represent the numbers. Thus, it would be a good idea to add a couple of functions, named getInt and getFloat, to the TerminalIO module, that would perform the interactive input of numeric values. Each of these functions, like getString, expects a string prompt as an argument. But they now must return an IO value of the relevant numeric type, rather than a string. This is demonstrated in the code for the function getFloat:

getFloat :: String -> IO Float
getFloat prompt = do
  inputString <- getString prompt
  return (read inputString :: Float)

The type signature of the getFloat function includes a return type of IO Float, which is just what you would expect. Note also how getFloat passes the buck to the useful getString function, already defined, which handles the details of the output prompt and the string input. All that getFloat does is to massage the input string into a Float for the use of the caller.
Now you can write typical interactive programs that process numeric inputs as well as text. For example, here is a program that computes and displays the tax for a mythical taxation system. The program accepts as inputs an income amount, a flat tax rate, a number of exemptions, and an individual exemption amount:

```
{-# File: TaxCode.hs #-}

module Main (main) where

import TerminalIO (getInt, getFloat)

main :: IO ()
main = do
    income <- getFloat "Enter your income: 
    exemptionAmt <- getFloat "Enter the exemption amount: 
    exemptions <- getInt "Enter the number of exemptions: 
    taxRate <- getInt "Enter your tax rate as a percent: 
    tax = income * fromIntegral taxRate / 100.0 - 
            exemptionAmount * fromIntegral exemptions
    putStrLn ("Your tax is " ++ show tax)

Note the use of the fromIntegral function to convert the values of taxRate and exemptions, which are of type Int, to a type that can be used with a floating-point number in arithmetic. Here is a test of the program in the GHCi:

*Main> main
Enter your income: 50000
Enter the exemption amount: 3500
Enter the number of exemptions: 2
Enter your tax rate as a percent: 25
Your tax is 5500.0
```

When you review the two programs listed in this section, you will see a common structure consisting of three parts:

1. a set of actions that obtain input data
2. a set of actions that process these data
3. a set of actions that output the results of the data processing

The first and third parts of these programs are considered impure code, meaning that they involve the use of I/O functions with side effects. The middle or data processing part, however, might just involve transforming the data with pure code, because no I/O operations occur there.

In any case, when the data processing part of a program becomes complex, it’s a good idea to isolate the functions and data types for that in separate modules. This design strategy not only helps to separate the pure code from the impure code in your programs, but also reflects the separation of concerns between the code that handles interactions with the external world and the code that merely processes data. The diagram in Figure 8 depicts this separation of concerns.
To illustrate this point with a simple example, consider a program that converts degrees Fahrenheit to degrees Celsius. The program consists of three modules, a top-level **Main** module, the **TerminalIO** module, and a data processing module named **Conversions**. The **Main** module defines the **main** function, which structures the I/O and data processing actions, as usual. The **Main** module relies upon a **TerminalIO** function for custom input. But after the input data are entered, they are sent off to a pure function defined in the **Conversions** module, which then returns the results to the **Main** module for output. Figure 9 shows the structure of this program.

Here is the code for the **Main** module and the **Conversions** module:

```haskell
{- File: TempConvert.hs -}
module Main (main) where

import Conversions (fahrToCelsius)
import TerminalIO (getFloat)

main :: IO ()
main = do
    degreesF <- getFloat "Enter the degrees Fahrenheit: "
    let degreesC = fahrToCelsius degreesF
    putStrLn "The degrees Celsius is "
    putStrLn (show degreesC)

{- File: Conversions.hs -}
module Conversions (celsiusToFahr, fahrToCelsius) where
```
```haskell
celsiusToFahr :: Float -> Float
celsiusToFahr degreesC = degreesC * 9 / 5 + 32

fahrToCelsius :: Float -> Float
fahrToCelsius degreesF = (degreesF - 32) * 5 / 9
```

Although this application might seem too simple to warrant separate modules, the practice is a good one to learn early, and will pay you big dividends as your programs grow more complex and sophisticated.

**Exercises**

In the guess the number game, a player thinks of a number between 1 and 100 and the other player repeatedly guesses the number, until the correct number is given. The guesser should be able to guess the number correctly with no more than approximately \(\log_2(100)\) guesses.

Write a program in Haskell in which the human user thinks of the number and the computer guesses it. After each incorrect guess, the user provides useful hints. Examples of useful hints would be "too small" and "too large." If the computer guesses correctly, the user enters "correct," whereupon the program displays the total number of guesses and halts.

Note that the computer should not have to emit more than the maximum number of guesses for this range of numbers. If that happens, the program should print an error message and halt.

**Text File Processing**

Text processing forms the basis for many applications, including word processing, email, web applications, and text messaging. In this section, you explore how to obtain input text from a text file and send output text to a text file.

The easiest way to perform I/O with text files is to use the built-in Haskell functions `readFile` and `writeFile`.

The `readFile` function expects the pathname of a text file as an argument. If the file exists, the function opens it for input, reads in all of its text, and retains this text in an action of type `IO String` for further access by your code. If you assume that a text file named `threelines.txt` exists in the current working directory, the next GHCi session shows some experimentation with the `readFile` function:

```haskell
Prelude> :type readFile
readFile :: FilePath -> IO String
Prelude> readFile "threelines.txt"
"There are three lines of text in this file."
Prelude> text <- readFile "threelines.txt"
Prelude> putStrLn text
There are three lines of text in this file.
```
Note that the GHCi implicitly extracts the text from the `readFile` function at the top level. This text shows two embedded newline characters. Note that `putStrLn` displays the text with the actual line breaks.

To create this file from a program, you can use the `writeFile` function. This function expects the file’s pathname and the output text as arguments. If the file does not exist, it is created; otherwise, the existing file is overwritten with the new text. Like the interactive output functions, `writeFile` returns an action of type `IO ()`. The next session shows the creation of the file used in the earlier session:

```
Prelude> :type writeFile
writeFile :: FilePath -> String -> IO ()
Prelude> let text = "There are three\nlines of text\nin this file."
Prelude> writeFile "threelines.txt" text
```

Armed with `readFile` and `writeFile`, you can write programs that input text, process it in some way, and output results, either to the terminal or to a text file.

One example application gathers and displays statistics about a text file – a line count, a word count, and a character count. The program prompts the user for the input file and then reads in its text. You can run the `lines` and `words` functions in the `Data.String` module to obtain lists of words and lines from this string, and then run the `length` function on these lists and on the string to obtain the required results. These statistics are suitably labeled and displayed in the terminal. Here is the code for this program:

```
{-# File: FileStats.hs #-}

module Main (main) where

import Data.String (lines, words)
import TerminalIO (getString)

main :: IO ()
main = do
  pathname <- getString "Enter the file name: 
  text <- readFile pathname
  let lineCount = length (lines text)
  let wordCount = length (words text)
  let charCount = length text
  putStrLn ("Line count: " ++ show lineCount)
  putStrLn ("Word count: " ++ show wordCount)
  putStrLn ("Character count: " ++ show charCount)
```

When run on its own source file, the program displays these results:

```
*Main> main
Enter the file name: FileStats.hs
Line count: 19
Word count: 72
Character count: 495
```
Earlier you developed a function named `numberLines` to number the lines in a piece of text. You can apply this function in a program that numbers the lines in a text file, as follows:

```haskell
{-
File: NumberLines.hs
-}

module Main (main) where

import Data.String (lines, unlines)
import TerminalIO (getString)

numberLines :: [Char] -> [Char]
numberLines text = unlines (helper (lines text) 1) where
    helper :: [[[Char]]] -> Int -> [[[Char]]]
    helper [] _ = []
    helper (x:xs) count =
        (show count ++ " " ++ x) : helper xs (count + 1)

main :: IO ()
main = do
    inPathName <- getString "Enter the input file name: "
    outPathName <- getString "Enter the input file name: "
    if inPathName == outPathName then do
        putStrLn "Error: files must have different names"
        return ()
    else do
        inputText <- readFile inPathName
        let outputText = numberLines inputText
        writeFile outPathName outputText
```

Note that the `numberLines` function is simply copied into the main module file, but it could just as easily be imported from its own module.

**Exercises**

A *concordance* is a program that computes and displays the number of instances of each word in a text file. Write a program that performs this task. The program should maintain an association list of words and their frequencies. As each word is obtained from the file, you remove its punctuation characters and convert it to uppercase. You then see if the word is already in the list. If it is not, you insert it with a frequency of 1. If the word is in the list, you replace its frequency with the incremented frequency. At the end of the input process, you sort the association list by key, and then output its contents in two columns.
Part VII. More Side Effects and Impurities: Random Numbers

As you saw in Part VI, having to deal with the effects of the external world of input and output data forces you to embed the functional model of computation in another model called the imperative model. The imperative model of computation tolerates side effects or changes of state in a program, which consists of a sequence of statements or commands. The types of side effects that you have seen thus far involve input and output operations, where output functions are called only for their side effects, and input functions return different values for the same arguments.

The need to perform interactive or file I/O is not the only source of side effects in computation. Although interactions with large and complex data structures can in principle be modeled as purely functional transformations of data into other data, it is convenient and often more efficient to treat these structures as objects whose states change between successive function applications. A thorough consideration of this type of programming in Haskell, which is not gentle, would require the introduction of Haskell’s monad construct. Unfortunately, such a discussion is beyond the scope of this book, although some good sources for this advanced topic are listed in the bibliography for the interested reader. Nevertheless, you will conclude this introduction to functional programming with a quite useful exploration of some applications that use random numbers, to get a feel for this type of programming.

Random Numbers

Random numbers are useful in applications that model real-world processes that exhibit some randomness or nondeterministic behavior. As you have seen, pure functions are thoroughly deterministic: they guarantee the same outcomes with the same arguments. For example, \( \text{sqrt} \ 4 \) always produces 2.0. For this reason, you could not define a pure function that simulates a coin toss, or chooses an alternative with a given probability, or selects an item from a random position is a list. You have seen how a Haskell programmer can add an imperative model of programming to support input and output operations; if you could use that model to define operations that generate random numbers and define other functions that use these numbers, you would be able to build applications to model nondeterministic real-world systems. In this section, you will learn how to define some impure functions to generate random numbers.

Your goal is to be able to use an impure function named \( \text{randomInt} \), which expects an integer as an argument. Its action is to generate a random number ranging from 0 through the argument number minus one. Here are some sample runs of the proposed function:

\[
\begin{align*}
\text{Prelude} & \> \text{:type randomInt} \\
\text{randomInt} & \text{:: Int} \rightarrow \text{IO Int} \\
\text{Prelude} & \> \text{do number} \leftarrow \text{randomInt} \ 2; \ \text{putStrLn (show number)} \\
& \quad 1 \\
\text{Prelude} & \> \text{do number} \leftarrow \text{randomInt} \ 2; \ \text{putStrLn (show number)} \\
& \quad 0 \\
\text{Prelude} & \> \text{do number} \leftarrow \text{randomInt} \ 10; \ \text{putStrLn (show number)} \\
& \quad 5 \\
\text{Prelude} & \> \text{do number} \leftarrow \text{randomInt} \ 10; \ \text{putStrLn (show number)} \\
& \quad 8
\end{align*}
\]
Prelude> do number <- randomInt 10; putStrLn (show number)
3
Prelude> do number <- randomInt 10; putStrLn (show number)
4

Note that the return type of the randomInt is **IO Int**, indicating an action from which you can extract an integer with the <- operation. Thus, your new function can be used only in the context of a do block, either in the GHCi or in other functions of type IO a. Also note that the function generates different values from the same arguments, so obviously it will have to rely upon something that changes its state between calls of the function.

Some random number generators start with a numeric seed value. Each time a new random number is called for, it is computed as a function of the seed, and then the seed is modified in some way so that the next number is a function of the new seed value. Unfortunately, this method causes the same sequence of numbers to be generated each time the application is run. A better random number generator relies on the system clock to provide the changes of state necessary to produce random numbers. Because each new run of an application starts at a different time, different sequences of random numbers are generated for each run. An initial seed can be derived from the current time at program startup, or each new random number can be derived directly from the current time.

A computer’s clock tracks the current time as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970. UTC is not a distinct time zone, but a standard adopted for synchronizing the world’s timing systems. However, UTC and GMT (Greenwich Mean Time) still share the same time in practice. Thus, each new reading of the computer’s clock is guaranteed to produce a different value.

Your task is to obtain a large integer from the current time on the system clock; you can then produce a number between 0 and a given upper bound minus one by taking the remainder of the large number and the upper bound. The two modules required to solve this problem are **Data.Time.Clock** and **Data.Time.Format**. You can import these modules into the GHCi as usual, and learn how to obtain a value from the system clock that you can use to generate a random number.

The **Data.Time.Clock.getCurrentTime** function obtains the current time from the system clock. Here is a sample run that shows the current time:

Prelude> import Data.Time.Clock
Prelude Data.Time.Format> getCurrentTime
2016-01-09 14:17:16.467388 UTC

Note that this function returns the UTC time with the date, hours in 24-hour format, minutes, seconds, and picoseconds (a picosecond is a trillionth of a second).

To massage the current time into a large integer that you can use in your random number function, you need to extract the picoseconds component. The function **formatTime** in
the `Data.Time.Format` module expects a time locale, a format string, and a time value as arguments, and returns a string representing the time in the given format. The function `defaultTimeLocale` gives the desired time locale information, as follows:

```
Prelude> defaultTimeLocale
TimeLocale {wDays = [('Sunday','Sun'),('Monday','Mon'),('Tuesday','Tue'),
('Wednesday','Wed'),('Thursday','Thu'),('Friday','Fri'),
('Saturday','Sat')], months = [('January','Jan'),('February','Feb'),
('March','Mar'),('April','Apr'),('May','May'),('June','Jun'),
('July','Jul'),('August','Aug'),('September','Sep'),('October','Oct'),
('November','Nov'),('December','Dec')], amPm = ('AM','PM'),
datetimeFmt = "%a %b %e %H:%M:%S %Z %Y",
dateFmt = "%m/%d/%y", timeFmt = "%H:%M:%S",
time12Fmt = "%I:%M:%S %p",
knownTimeZones = [UT,GMT,EST,EDT,CST,CDT,MST,MDT,PST,PDT]}
```

When called with this time locale, the format string "%H:%M:%S:%q" and the current time, the `formatTime` function returns a string containing the time components in hours, minutes, seconds, and picoseconds:

```
Prelude> do {time <- getCurrentTime;
return (formatTime defaultTimeLocale "%H:%M:%S:%q" time)}
"14:33:28:953837000000"
```

Using just the format string "%q", you get a string containing just the picoseconds:

```
Prelude> do {time <- getCurrentTime;
return (formatTime defaultTimeLocale "%q" time)}
"365663000000"
```

You then take the first six digits of this string and convert it to an `Int` with the `read` function:

```
Prelude> do {time <- getCurrentTime;
return (read $ take 6 $ formatTime defaultTimeLocale "%q" time :: Int)}
516411
```

Now you have a large integer, generated from a tick of the system clock, from which you can generate a random number. All that remains is to take the remainder of this integer and the upper bound of the range plus one. Here is the code for the `randomInt` function and a tester function in the module `MyRandom`:

```
module MyRandom where

import Data.Time.Clock
import Data.Time.Format

-- Impure function returns a random number between
-- 0 and n - 1, inclusive.
randomInt :: Int -> IO Int
randomInt n = do
```

```
time <- getCurrentTime
return ((`rem` n) $ read $ take 6 $
        formatTime defaultTimeLocale "%q" time)

-- Prints random integers between 0 and n - 1, for
-- a given number of iterations.
testRandomInt :: Int -> Int -> IO ()
testRandomInt n iterations = do
  if iterations == 0 then
    return ()
  else do
    r <- randomInt n
    putStrLn (show r)
    testRandomInt n (iterations - 1)

Note that the randomInt function returns an action of type IO Int, while the
testRandomInt function, which is run just for side effects, returns type IO (). A
sample run of the testRandomInt function shows the expected behavior:

Prelude> testRandomInt 2 5      -- 5 coin tosses
  1
  0
  1
  0
  0
Prelude> testRandomInt 10 5     -- 5 picks between 0 and 9
  8
  7
  6
  0
  3

You will put your random number generator to work in a couple of applications that
conclude this book; but first, some exercise.

Exercises
1. Modify the guessing game program from an earlier set of exercises by having the
   computer think of a number and the user supply the guesses. The computer’s number
   will be a random number between 1 and 100. The computer prompts the user for her
guess, which is input as a number. The computer then compares this guess to its own
number and outputs a clue (“too large” or “too small”) if the guess is incorrect. The
computer also tracks the number of the user’s guesses and does not allow the game to
continue past the maximum number of guesses needed.
2. Add a function pickRandom to the MyRandom module. This function expects a
   nonempty list as an argument and returns at item at a random position.
Generating Sentences from a Grammar

All languages, whether they are natural, like English and Spanish, or formal, like Java and Haskell, are defined by a vocabulary and a grammar. A vocabulary is the set of words available to users of a language, and a grammar is a set of rules for forming syntactically correct sentences from those words. In this section, you will learn how to model a vocabulary and grammar for a tiny subset of English; you can then use this information to generate random sentences in the language.

Grammars in any language reflect its phrase structure. The top-level phrase is the sentence. A sentence in turn might consist of other phrases, such as a noun phrase followed by a verb phrase. Certain phrases, such as the noun phrase, “bottom out” at parts of speech, such as an article and a noun. A part of speech is just a name for a member of a subset of words in the vocabulary. For example, the set of nouns might include “boy,” “girl,” “ball,” and “bat.” Other typical parts of speech are the verb, the preposition, the adjective, the adverb, and the conjunction. Other phrases might be the propositional phrase and the dependent clause.

At a certain point in your education, you might have learned how to diagram the grammatical structure of a given sentence. This diagram usually is a tree-like structure, whose items are labeled by the names of phrases, parts of speech, and words in the sentence. Figure 10 shows a phrase structure diagram of the sentence, “The girl hit the boy with a bat.”

FIGURE 10
A phrase structure diagram of a sentence

There are many ways to specify a grammar, and here you will use a notation called Extended Backus-Naur form (the same Backus discussed in the book’s Introduction). In this notation, each grammar rule is an equation, with the name of a phrase or part of speech on the left, and its definition on the right. A definition specifies the sequence of phrases and/or parts of speech that constitute the given phrase. Some definitions might include notation that indicates optional items; for example, a noun phrase might or might
not be modified by a prepositional phrase. Other definitions might need notation to indicate zero or more items; for example, a sequence of adjectives might modify a noun.

Aside from these features, the Extended Backus-Naur form uses the following conventions:

- All words in the vocabulary, which are also called *terminal symbols*, appear in quotation marks.
- All names of phrases, which are also called *nonterminal symbols*, are spelled without quotation marks.
- Several symbols, which are also called *metasymbols*, are used to indicate options, iteration, and so forth. They are `=`, `{}`, `[]`, `()`, and `|`. Their meanings are listed in Table 9.

<table>
<thead>
<tr>
<th>Metasymbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>Means is defined as</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>[]</code></td>
<td>Encloses an optional set of items</td>
</tr>
<tr>
<td><code>{}</code></td>
<td>Encloses a set of items that can repeat zero or more times</td>
</tr>
<tr>
<td><code>()</code></td>
<td>Groups together required items</td>
</tr>
</tbody>
</table>

Given these three sets of symbols, the grammar rule

```
sentence = nounPhrase verbPhrase
```

can be read as “A sentence is a noun phrase followed by a verb phrase,” and the rule

```
nounPhrase = article { adjective } noun
```

can be read as “A noun phrase is an article, followed by zero or more adjectives, followed by a noun.”

The parts of speech, which are also nonterminal symbols, are defined as sets of optional terminal symbols. For example, the rule

```
article = "a" | "the"
```

which says, “An article is either the word ‘a’ or the word ‘the’,” specifies the complete set of articles in the vocabulary of this language.

Armed with this notation, you can construct a complete grammar for a tiny subset of English, as follows:

```
sentence = nounPhrase verbPhrase
```
verbPhrase = verb nounPhrase prepositionalPhrase

nounPhrase = article noun

prepositionalPhrase = preposition nounPhrase

The rules for defining the parts of speech article, noun, verb, and preposition are omitted for now. You will see shortly how these are represented as components of a vocabulary in a Haskell program.

You will note that the each of the nonterminal symbols on the left side of a grammar rule looks suspiciously like it could be the name of a function. In fact, it will be a fairly easy and straightforward process to translate each of these rules to a function definition that generates a random phrase in the given language.

First, you represent the vocabulary as lists of words named by the parts of speech to which they belong:

```
-- Vocabulary consists of nouns, verbs, prepositions, -- and articles
nouns :: [String]
nouns = ["bat", "boy", "girl", "dog", "cat", "chair",
          "fence", "table", "computer", "cake", "field"]

verbs :: [String]
verbs = ["hit", "threw", "pushed", "ate", "dragged",
          "jumped"]

prepositions :: [String]
prepositions = ["with", "to", "from", "on", "below",
                "above", "beside"]

articles :: [String]
articles = ["a", "the"]
```

Then for each rule of the grammar, you define a function whose name is the nonterminal symbol on the left side of the rule. The function expects no arguments and returns an action of type IO String. All of these functions must be impure, because they ultimately depend upon selecting words at random from the vocabulary. Each function then builds a string from the results obtained by calling the functions corresponding to the nonterminal symbols on the right side of its rule, or picking a word at random from the vocabulary when the nonterminal symbol is a part of speech. Here is the code for the definitions of the four functions required by your grammar:

```
-- Syntax of sentences and other phrases

-- sentence = nounphrase verbphrase
sentence :: IO String
sentence = do
    np <- nounPhrase
    vp <- verbPhrase
    return (np ++ " " ++ vp)
-- nounphrase = article noun
nounPhrase :: IO String
nounPhrase = do
    article <- pickRandom articles
    noun <- pickRandom nouns
    return (article ++ " " ++ noun)

-- verbphrase = verb nounphrase prepositionalphrase
verbPhrase :: IO String
verbPhrase = do
    verb <- pickRandom verbs
    np <- nounPhrase
    pp <- prepositionalPhrase
    return (verb ++ " " ++ np ++ " " ++ pp)

-- prepositionalphrase = preposition nounphrase
prepositionalPhrase :: IO String
prepositionalPhrase = do
    prep <- pickRandom prepositions
    np <- nounPhrase
    return (prep ++ " " ++ np)

Note that you use the pickRandom function defined in the exercises of the last section to obtain a random word of a given part of speech from the vocabulary. Note also that you must be careful to use the do block and the <- operator to obtain a string from each function called within a definition. Finally, you must make a function’s string result available for further use by embedding it in a return expression.

You can load this module into the GHCi and test any of the functions at the top level. But you might also include a tester function that can generate and display a certain number of sentences, as follows:

testGenerator :: Int -> IO ()
testGenerator iterations = do
    if iterations == 0 then
        return ()
    else do
        s <- sentence
        putStrLn s
        testGenerator (iterations - 1)

Here is a sample run of this function to generate 10 grammatically correct but somewhat silly sentences:

Prelude> testGenerator 10
a fence hit the bat from the chair
the girl dragged the chair above a cake
the cat pushed the computer to the bat
a fence pushed the cake to a girl
the dog hit the fence above a cake
the computer threw a bat with the dog
a fence hit a cake beside a boy
a cat threw a computer above the bat
The syntax and semantics of the impure functions takes a while to get used to. But it is fairly easy to extend this grammar to cover other types of phrases and add words to the vocabulary to make your random sentences more realistic.

Exercises
1. Modify the rule for verb phrases so that the prepositional phrase is optional. The `verbPhrase` function should include a prepositional phrase about ¼ of the time.
2. Add new variables for lists of adjectives and conjunctions to the sentence generator program.
3. Noun phrases can be modified by multiple adjectives (“The little red ball . . .”) and sentences can have more than one independent clause (“The girl threw a pitch and the boy hit the ball”). Add the following rules to the grammar and modify your functions to support these rules (items enclosed in {} may be repeated zero or more times):

```haskell
sentence = nounPhrase verbPhrase {conjunction sentence}

nounPhrase = article {adjective} noun
```

Non-Directive Psychotherapy
At this point in the book, you might need a visit to your favorite psychotherapist, either to discuss a possible career change or to help recover from cognitive overload. In this section, you develop a program that simulates a therapist whom you can engage in such conversations.

In the early 1960s, the M.I.T computer scientist Joseph Weizenbaum developed a famous program called Doctor that could converse with the computer user, mimicking a nondirective style of psychotherapy. The doctor in this kind of therapy is essentially a good listener who responds to the patient’s statements by rephrasing them or indirectly asking for more information. Here is a sample run of this program in the GHCi:

```haskell
Prelude> main
Hello, how are you today?

>>>
My teacher always plays favorites

Why do you say that your teacher always plays favorites?

>>> She always calls on the girls in the class

Please tell me more.

>>>
```

As you can see, the doctor displays a greeting at program startup, followed by a >>> prompt for your input. In its simplest form, the doctor responds to your inputs in one of two ways:

1. With a randomly chosen hedge, such as “Please tell more.”
2. By changing some key words in your input string and appending the result to a randomly chosen qualifier. Thus, to “my teacher always plays favorites,” the doctor might reply, “Why do you say that your teacher always plays favorites?”

The conversation continues in this way until you enter the word “quit.”

Two of the data structures for the program are a list of hedges and a list of qualifiers. The remaining data structure is an association list of key words and their replacements. These associations represent mappings from first-person pronouns, such as “I” and “my,” to second-person pronouns, such as “you” and “your.” All three data structures are defined at the top level of the module, so they are set up as the module is loaded. Their names will be visible, without being passed as arguments, to any of the functions that need to access them. Here is the code for these items, as well as the module’s imports:

```haskell
module Main (main) where

import MyRandom (randomInt, pickRandom)
import TerminalIO (getString)
import Data.String (words, unwords)
import Data.Char (toUpper)

type Phrases = [String]
type AssociationList = [(String,String)]

hedges :: Phrases
hedges = ["Please tell me more.", "Please go on.",
         "Many of my clients say that."]

qualifiers :: Phrases
qualifiers = ["Why do you say that ",
              "You seem to think that ",
              "Did I just hear you say that "]

replacements :: AssociationList
replacements = [("I","you"), ("ME","you"), ("MY","your"),
                ("WE","you"), ("US","you")]
```

The rest of the program consists of four functions, named `main`, `driverLoop`, `reply`, and `changePerson`. Their interactions are shown in the structure chart in Figure 11.
Their responsibilities and Haskell definitions are discussed in what follows. Note that all but one of the functions is impure, because they use I/O operations or resources for random numbers.

The `main` function displays the doctor’s greeting, calls the `driverLoop` function to begin the conversation, and displays a farewell message when the conversation ends:

```haskell
driverLoop :: IO ()
driverLoop = do
  sentence <- getString(">>> ")
  if makeUppercase sentence == "QUIT" then
    return ()
  else do
    doctorReply <- reply sentence
    putStrLn (doctorReply ++ ",
    driverLoop
```
if chance == 0 then do
  hedge <- pickRandom hedges
  return hedge
else do
  let newSentence = changePerson sentence
  qualifier <- pickRandom qualifiers
  return (qualifier ++ newSentence ++ "?")

The changePerson function creates a list of the words from its input string, looks up each of these words in the replacements list, and substitutes the replacement for the word if that word’s replacement is in the replacements list. The function then converts the finished list of words back to a string to be returned to its caller:

changePerson :: String -> String
changePerson sentence =
  unwords $ map myLookup $ words sentence
  where
    myLookup :: String -> String
    myLookup word =
      maybe word id (lookup (makeUppercase word)
        replacements)

Note the use of the locally defined function myLookup in the definition of the changePerson function. This helper function standardizes the case of the input word so it can be found in the replacements list. But the lookup function that does the search returns a value of type Maybe. As you saw earlier, you can extract the word if it was found with a pattern match, but this involves an if/else expression within your code. To simplify the process, you use instead the built-in maybe function. Here is the type of this function:

Prelude> :type maybe
maybe :: b -> (a -> b) -> Maybe a -> b

Note that there are three arguments and one returned value. If the third argument is the value Nothing, then the first argument is returned as a default value. Otherwise, the third argument is Just x, so the second argument, which is a function, is applied to x to produce the value returned. For your purposes in the current program, you want the maybe function to give you either the default value or the value embedded in the Maybe value. You can pass maybe the built-in id function, which simply returns its argument, as follows:

Prelude> maybe "A default value" id Nothing
"A default value"
maybe "A default value" id (Just "Something")
"Something"

This concludes your introductory exploration of functional programming in Haskell. I hope that you have enjoyed your trip, and that you have many more exciting journeys ahead!
Exercises

1. When the patient addresses the doctor personally, the persons are not changed properly in the doctor's reply. Test the program with "you are not a helpful therapist" to see this. Fix this problem by repairing the dictionary of replacements.

2. Conversations often shift focus to earlier topics. Modify the doctor program to support this capability. Add each patient input to a history list. Then, occasionally choose an element at random from this list, change persons, and prepend the qualifier "Earlier you said that " to this reply. Be sure that the history list holds at least three entries before using it and that you add inputs to the history list after replies are generated. Hint: the main function creates an empty history list and passes it as an argument to the driverLoop function. This function passes the history list to the reply function, but adds the patient’s current input after reply is called.
Bibliography
There are a number of excellent books on functional programming and on functional programming with Haskell. The two books that we have found most useful follow.

O’Sullivan, Bryan, Goerzen, John, and Stewart, Don, *Real World Haskell* (O’Reilly, 2009). This book takes a cookbook approach to programming with the many Haskell APIs for applications that use JSON, binary files, databases, web clients, graphical user interfaces, networks, and concurrency.


For those who wish to explore programming in a non-functional language such as Java or Python, the following books provide some easygoing introductions.
