Haskell Programming

from first principles

Christopher Allen

Julie Moronuki

Pure functional programming without fear or frustration

0.1 Hello reader!

This is a sample of our third chapter on strings and printing. We've also included part of the front matter in the hopes that will explain some of how we're approaching teaching Haskell. We hope it gives you some idea as to what our "style" is like, although there might be some stuff that may not make sense without the rest of the introduction. If you have any questions, please see the book website at `http://haskellbook.com` for support information.

When to expect a release

We will do a release approximately once every month.

What we've got so far and the table of contents

Chapters available

12. Algebraic datatypes
13. (Surprise chapter)
14. Building projects in Haskell

On deck

Defined as being >50% completed

15. Testing
16. Monoid, Semigroup
17. Functors
18. Applicative
19. Monad (SPOOKY? No.)
20. Foldable/Traversable
21. Reader/State

Still being written

22. Parsers
23. Monad Transformers
24. Non-strictness and efficiency
25. Commonly used data structures
26. (Four final chapters TBA)

Does the material we cover sound like what the doctor ordered? Head to our gumroad page at `https://gumroad.com/l/haskellbook` to get in our early access period.

0.2 Introduction

Welcome to a new way to learn Haskell. Perhaps you are coming to this book frustrated by previous attempts to learn Haskell. Perhaps you have only the faintest notion of what Haskell is. Perhaps you are coming here because you are not convinced that anything will ever be better than Common Lisp/Scala/Ruby/whatever language you love, and you want to argue with us. Perhaps you were just looking for the 18 billionth *(n.b.: this number may be inaccurate)* monad tutorial, certain that this time around you will understand monads once and for all. Whatever your situation, welcome and read on! It is our goal here to make Haskell as clear, painless, and practical as we can, no matter what prior experiences you're bringing to the table.

Why Haskell

If you are new to programming entirely, Haskell is a great first language. You may have noticed the trend of "Functional Programming in [Imperative Language]" books and tutorials and learning Haskell gets right to the heart of what functional programming is. Languages such as Java are gradually adopting functional concepts, but most such languages were not designed to be functional languages. We would not encourage you to learn Haskell as an *only* language, but because Haskell is a pure functional language, it is a fertile environment for mastering functional programming. That way of thinking and problem solving is useful, no matter what other languages you might know or learn.

If you are already a programmer, writing Haskell code may seem to be more difficult up front, not just because of the hassle of learning a language that is syntactically and conceptually different from a language you already know, but also because of features such as strong typing that enforce some discipline in how you write your code. Without wanting to trivialize the learning curve, we would argue that part of being a professional means accepting that your field will change as new information and research demands it.

It also means confronting human frailty. In software this means that we *cannot* track all relevant metadata about our programs in our heads ourselves. We have limited space for our working memory, and using it up for

anything a computer can do for us is counter-productive. We don't write Haskell because we're geniuses — we use tools like Haskell because we're not geniuses and it helps us. Good tools like Haskell enable us to work faster, make fewer mistakes, and have more information about what our code is supposed to do as we read it.

We use Haskell because it is easier (over the long run) and enables us to do a better job. That's it. There's a ramp-up required in order to get started, but that can be ameliorated with patience and a willingness to work through exercises.

What is Haskell for?

Haskell is a general purpose, functional programming language. It's not scoped to a particular problem set. It's applicable virtually anywhere one would use a program to solve a problem, save for some specific embedded applications. If you could write software to solve a problem, you could probably use Haskell.

Haskell's ecosystem has an array of well-developed libraries and tools. Hoogle is a search tool that allows you to search for the function you want by type signature. It has a sophisticated structural understanding of types and can match on more than just syntax. Libraries such as Yesod and Scotty allow you to build web applications quickly, each addressing a different niche of web development. Aeson is popular and in wide use in the Haskell community for processing JSON data which is currently the lingua franca for data serialization and transmission on the web. Gloss is popular for rendering 2d vector graphics. Xmonad is a tiled window manager for Linux/X11, written in Haskell and popular with Haskell users and non-Haskellers. Many more people use pandoc which is as close to a universal document conversion and processing tool as currently exists, and is also written in Haskell.

Haskell is used in industrial settings like oil & gas control systems, data anonymization, mobile applications for iOS and Android, embedded software development, REST APIs, data processing at scale, and desktop applications. Entire financial front offices have used Haskell to replace everything from C++ trading applications to apps running in Excel, and even some indie game developers are using Haskell with functional reactive programming

libraries to make their projects.

OK, but I was just looking for a monad tutorial...

We encourage you to forget what you might already know about programming and come at this course in Haskell with a beginner's mindset. Make yourself an empty vessel, ready to let the types flow through you.

If you are an experienced programmer, learning Haskell is more like learning to program all over again. Getting somebody from Python to JavaScript is comparatively trivial. They share very similar semantics, structure, type system (none, for all intents and purposes), and problem-solving patterns (e.g., representing everything as maps, loops, etc.). Haskell, for most who are already comfortable with an imperative or untyped programming language, will impose previously unknown problem-solving processes on the learner. This makes it harder to learn not because it is intrinsically harder, but because most people who have learned at least a couple of programming languages are accustomed to the process being trivial, and their expectations have been set in a way that lends itself to burnout and failure.

If you are learning Haskell as a first language, you may have experienced a specific problem with the existing Haskell materials and explanations: they assume a certain level of background with programming, so they frequently explain Haskell concepts in terms, by analogy or by contrast, of programming concepts from other languages. This is obviously confusing for the student who doesn't know those other languages, but we posit that it is just as unhelpful for experienced programmers. Most attempts to compare Haskell with other languages only lead to a superficial understanding of the Haskell concepts, and making analogies to loops and other such constructs can lead to bad intuitions about how Haskell code works. For all of these reasons, we have tried to avoid relying on knowledge of other programming languages. Just as you can't achieve fluency in a human language so long as you are still attempting direct translations of concepts and structures from your native language to the target language, it's best to learn to understand Haskell on its own terms.

This book is not something you want to just leaf through the first time you read it. It is more of a course than a book, something to be worked

through. There are exercises sprinkled liberally throughout the book; we encourage you to do all of them, even when they seem simple. Those exercises are where the majority of your epiphanies will come from. No amount of chattering, no matter how well structured and suited to your temperament, will be as effective as *doing the work*. We do not recommend that you pass from one section of the book to the next without doing at least a few of the exercises. If you do get to a later chapter and find you did not understand a concept or structure well enough, you can always return to an earlier chapter and do more exercises until you understand it. The Freenode IRC channel `#haskell-beginners` has teachers who will be glad to help you as well, and they especially welcome questions regarding specific problems that you are trying to solve.[1]

We believe that spaced repetition and iterative deepening are effective strategies for learning, and the structure of the book reflects this. You may notice we mention something only briefly at first, then return to it over and over. As your experience with Haskell deepens, you have a base from which to move to a deeper level of understanding. Try not to worry that you don't understand something completely the first time we mention it. By moving through the exercises and returning to concepts, you can develop a solid intuition for the functional style of programming.

The exercises in the first few chapters are designed to rapidly familiarize you with basic Haskell syntax and type signatures, but you should expect exercises to grow more challenging in each successive chapter. Where possible, reason through the code samples and exercises in your head first, then type them out — either into the REPL[2] or into a source file — and check to see if you were right. This will maximize your ability to understand and reason about Haskell. Later exercises may be difficult. If you get stuck on an exercise for an extended period of time, proceed and return to it at a later date.

[1] Freenode IRC (Internet Relay Chat) is a network of channels for textual chat. There are other IRC networks around, as well as other group chat platforms, but the Freenode IRC channels for Haskell are popular meeting places for the Haskell community. There are several ways to access Freenode IRC if you're interested in getting to know the community in their natural habitat.

[2] A REPL, short for Read-Eval-Print-Loop, is a program you use to type code in and see how the expressions evaluate. We'll explain this when we introduce GHCi.

You are not a Spartan warrior — you do not need to come back either with your shield or on it. Returning later to investigate things more deeply is an efficient technique, not a failure.

What's in this book?

We cover a mix of practical and abstract matters required to use Haskell for a wide variety of projects. Chris's experience is principally with production backend systems and frontend web applications. Julie is a linguist and teacher by training and education, and learning Haskell was her first experience with computer programming. The educational priorities of this book are biased by those experiences. Our goal is to help you not just write typesafe functional code but to understand it on a deep enough level that you can go from here to more advanced Haskell projects in a variety of ways, depending on your own interests and priorities.

One of the most common responses we hear from people who have been trying to learn Haskell is, "OK, I've worked through this book, but I still don't know how to actually write anything of my own." We've combined practical exercises with principled explanations in order to alleviate that problem.

Each chapter focuses on different aspects of a particular topic. We start with a short introduction to the lambda calculus. What does this have to do with programming? All modern functional languages are based on the lambda calculus, and a passing familiarity with it will help you down the road with Haskell. If you've understood the lambda calculus, understanding the feature of Haskell known as "currying" will be a breeze, for example.

The next few chapters cover basic expressions and functions in Haskell, some simple operations with strings (text), and a few essential types. You may feel a strong temptation, especially if you have programmed previously, to glance at those chapters, decide they are too easy and skip them. *Please do not do this.* Even if those first chapters are covering concepts you're familiar with, it's important to spend time getting comfortable with Haskell's rather terse syntax, making sure you understand the difference between working in the REPL and working in source files, and becoming familiar with the compiler's sometimes quirky error messages. Certainly you may

work quickly through those chapters — just don't skip them.

From there, we build both outward and upward so that your understanding of Haskell both broadens and deepens. When you finish this book, you will not just know what monads are, you will know how to use them effectively in your own programs and understand the underlying algebra involved. We promise — you will. We only ask that you do not go on to write a monad tutorial on your blog that explains how monads are really just like jalapeno poppers.

In each chapter you can expect:

- additions to your vocabulary of standard functions;
- syntactic patterns that build on each other;
- theoretical foundations so you understand how Haskell works;
- illustrative examples of how to read Haskell code;
- step-by-step demonstrations of how to write your own functions;
- explanations of how to read common error messages and how to avoid those errors;
- exercises of varying difficulty sprinkled throughout;
- definitions of important terms.

We have put the definitions at the end of each chapter. Each term is, of course, defined within the body of the chapter, but we added separate definitions at the end as a point of review. If you've taken some time off between one chapter and the next, the definitions can remind you of what you have already learned, and, of course, they may be referred to any time you need a refresher.

A few words about working environments

This book does not offer much instruction on using the terminal and text editor. The instructions provided assume you know how to find your way

around your terminal and understand how to do simple tasks like make a directory or open a file. Due to the number of text editors available, we do not provide specific instructions for any of them.

If you are new to programming and don't know what to use, consider using an editor that is unobtrusive, uses the Common User Access bindings most computer users are accustomed to, and doesn't force you to learn much that is new (assuming you're already used to using commands like ctrl-c to copy), such as Gedit or Sublime Text. When you're initially learning to program, it's better to focus on learning Haskell rather than struggling with a new text editor at the same time. However, if you believe programming will be a long-term occupuation or preoccupation of yours, you may want to move to text editors that can grow with you over time such as Emacs or Vim.

0.3 A few words about the examples and exercises

We have tried to include a variety of examples and exercises in each chapter. While we have made every effort to include only exercises that serve a clear pedagogical purpose, we recognize that not all individuals enjoy or learn as much from every type of demonstration or exercise. Also, since our readers will necessarily come to the book with different backgrounds, some exercises may seem too easy or difficult to you but be just right for someone else. Do your best to work through as many exercises as seems practical for you. But if you skip all the types and typeclasses exercises and then find yourself confused when we get to Monoid, by all means, come back and do more exercises until you understand.

Here are a few things to keep in mind to get the most out of them:

- Examples are usually designed to demonstrate, with real code, what we've just talked or are about to talk about in further detail.
- You are intended to type *all* of the examples into the REPL or a file and load them. We *strongly* encourage you to attempt to modify the example and play with the code after you've made it work. Forming hypotheses about what effect changes will have and verifying them is critical!
- Sometimes the examples are designed intentionally to be broken. Check surrounding prose if you're confused by an unexpected error as we will not show you code that doesn't work without commenting on the breakage. If it's still broken and it's not supposed to be, you should start checking your syntax for errors.
- Not every example is designed to be entered into the REPL. Not every example is designed to be entered into a file. We explain the syntactic differences between files and REPL expressions. You are expected to perform the translation between the two yourself after it has been explained. This is so you are accustomed to working with code in an interactive manner by the time you finish the book.

- Exercises in the body of the chapter are usually targeted to the information that was covered in the immediately preceding section(s).
- Exercises at the end of the chapter generally include some review questions covering material from previous chapters and are more or less ordered from easiest to most challenging. Your mileage may vary.
- Even exercises that seem easy can increase your fluency in a topic. We do not fetishize difficulty for difficulty's sake. We just want you to understand the topics as well as possible. That can mean coming at the same problem from different angles.
- We ask you to write and then rewrite (using different syntax) a lot of functions. Few problems have only one possible solution, and solving the same problem in different ways increases your fluency and comfort with the way Haskell works (its syntax, its semantics, and in some cases, its evaluation order).
- Do not feel obligated to do all the exercises in a single sitting or even in a first pass through the chapter. In fact, spaced repetition is generally a more effective strategy.
- Some exercises, particularly in the earlier chapters, may seem very contrived. Well, they are. But they are contrived to pinpoint certain lessons. As the book goes on and you have more Haskell under your belt, the exercises become less contrived and more like "real Haskell."
- It is better to type the code examples and exercises yourself rather than copy and paste. Typing it out yourself makes you pay more attention to it, so you ultimately learn much more from doing so.
- We recommend you move away from typing code examples and exercises directly into GHCi sooner rather than later and develop the habit of working in source files. It isn't just because the syntax can become unwieldy in the REPL. Editing and modifying code, as you will be doing a lot as you rework exercises, is easier and more practical in a source file. You will still load your code into the REPL from the file to run it, and we'll cover how to do this in the appropriate chapter.

- Another benefit to writing code in a source file and then loading it into the REPL is that you can write comments about the process you went through in solving a problem. Writing out your own thought process can clarify your thoughts and make the solving of similar problems easier. At the very least, you can refer back to your comments and learn from yourself.

- Sometimes we intentionally underspecify function definitions. You'll commonly see things like:

  ```
  f = undefined
  ```

  Even when *f* will probably take named arguments in your implementation, we're not going to name them for you. Nobody will scaffold your code for you in your future projects, so don't expect this book to either.

Chapter 2

Hello, Haskell!

Basic expressions and functions

> Functions are beacons of
> constancy in a sea of turmoil.
>
> Mike Hammond

2.1 Hello, Haskell

Welcome to your first step in learning Haskell. Before you begin with the main course of this book, you will need to install the tools necessary to work with the language in order to complete the exercises as you work through the book. You can find the installation instructions online at `https://github.com/bitemyapp/learnhaskell`. If you wish, you can complete the installation and practice only for GHCi now and wait to install Cabal, but you will need Cabal up and running for the chapter about building projects. The rest of this chapter will assume that you have completed the installation and are ready to begin working.

In this chapter, you will

- use Haskell code in the interactive environment and also from source files;
- understand the building blocks of Haskell: expressions and functions;
- learn some features of Haskell syntax and conventions of good Haskell style;
- modify simple functions.

2.2 Interacting with Haskell code

Haskell offers two primary ways of writing and compiling code. The first is inputting it directly into the interactive environment known as GHCi, or the REPL. The second is typing it into a text editor, saving, and then loading that source file into GHCi. This section offers an introduction to each method.

Using the REPL

REPL is an acronym short for Read-Eval-Print Loop. REPLs are interactive programming environments where you can input code, have it evaluated

by the language implementation, and see the result. They originated with `Lisp` but are now common to modern programming languages including Haskell.

Assuming you've completed your installation, you should be able to open your terminal or command prompt, enter `ghci`, hit enter, and see something like the following:

```
GHCi, version 7.8.3
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

It might not be exactly the same, depending on your configuration, but it should be very similar. If you are just downloading GHC for the first time, you may have the newest release number, which is 7.10.1. The newest version includes some important changes which we will cover in due time.

Now try entering some simple arithmetic at your prompt:

```
Prelude> 2 + 2
4
Prelude> 7 < 9
True
Prelude> 10 ^ 2
100
```

If you can enter simple equations at the prompt and get the expected results, congratulations — you are now a functional programmer! More to the point, your REPL is working well and you are ready to proceed.

To exit GHCi, use the command `:quit` or `:q`.

Abbreviating GHCi commands Throughout the book, we'll be using GHCi commands, such as `:quit` and `:info` in the REPL. We will present them in the text spelled out, but they can generally be abbreviated to just

the colon and the first letter. That is, `:quit` becomes `:q`, `:info` becomes `:i` and so forth. It's good to type the word out the first few times you use it, to help you remember what the abbreviation stands for, but as you remember the commands, we will start abbreviating them and you will, too.

Saving scripts and loading them

As nice as REPLs are, usually you want to store code in a file so you can build it incrementally. Almost all nontrivial programming you do will involve editing libraries or applications made of nested directories containing files with Haskell code in them. The basic process is to have the code and imports (more on that later) in a file, load it into the REPL, and interact with it there as you're building, modifying, and testing it.

You'll need a file named `test.hs`. The `.hs` file extension denotes a Haskell source code file. Depending on your setup and the workflow you're comfortable with, you can make a file by that name and then open it in your text editor or you can open your text editor, open a new file, and then save the file with that file name.

Then enter the following code into the file and save it:

```
sayHello :: String -> IO ()
sayHello x = putStrLn ("Hello, " ++ x ++ "!")
```

Here, `::` is a way to write down a type signature. You can think of it as saying, "has the type." So, `sayHello` has the type `String -> IO ()`. These first chapters are focused on syntax, so if you don't understand what types or type signatures are, that's okay — we will explain them soon. For now, keep going.

Then in the same directory where you've stored your `test.hs` file, open your `ghci` REPL and do the following:

```
Prelude> :load test.hs
Prelude> sayHello "Tina"
Hello, Tina!
```

```
Prelude>
```

After using `:load` to load your `test.hs`, the `sayHello` function is visible in the REPL and you can pass it a string argument, such as "Tina" (note the quotation marks), and see the output.

Note that we used the command `:load` to load the file. Special commands that only GHCi understands begin with the `:` character. `:load` is *not* Haskell code; it's just a GHCi feature. We will see more of these commands throughout the book.

2.3 Understanding expressions

Everything in Haskell is an expression or declaration. Expressions may be values, combinations of values, and/or functions applied to values. Expressions evaluate to a result. In the case of a literal value, the evaluation is trivial as it only evaluates to itself. In the case of an arithmetic equation, the evaluation process is the process of computing the operator and its arguments, as you might expect. But, even though not all of your programs will be about doing arithmetic, all of Haskell's expressions work in a similar way, evaluating to a result in a predictable, transparent manner. Expressions are the building blocks of our programs, and programs themselves are one big expression made of smaller expressions. Declarations we'll cover more later, but it suffices to say for now that they are top-level bindings which lets us give names to expressions which we can use to refer to them multiple times without copying and pasting the expressions.

The following are all expressions:

```
1
1 + 1
"Icarus"
```

Each can be examined in the GHCi REPL where you can enter the code at the prompt, then hit 'enter' to see the result of evaluating the expression. The numeric value 1, for example, has no further reduction step, so it stands

for itself. If you haven't already, open up your terminal and at the prompt, enter `ghci` to get your REPL going and start following along with the code examples.

When we enter this into GHCi:

```
Prelude> 1
1
```

We see 1 printed because it cannot be reduced any further.

In the next example, GHCi reduces the expression `1 + 2` to 3, then prints the number 3. The reduction terminates with the value 3 because there are no more terms to evaluate:

```
Prelude> 1 + 2
3
```

Expressions can be nested in numbers limited only by our willingness to take the time to write them down, much like in arithmetic:

```
Prelude> (1 + 2) * 3
9
Prelude> ((1 + 2) * 3) + 100
109
```

You can keep expanding on this, nesting as many expressions as you'd like and evaluating them. But, we don't have to limit ourselves to expressions such as these.

Normal form We say that expressions are in *normal form* when there are no more evaluation steps that can be taken, or, put differently, when they've reached an irreducible form. The normal form of `1 + 1` is 2. Why? Because the expression `1 + 1` can be evaluated or reduced by applying the addition operator to the two arguments. In other words, `1 + 1` is a reducible expression, while 2 is an expression but is no longer reducible — it can't

evaluate into anything other than itself. Reducible expressions are also called *redexes*. While we will generally refer to this process as evaluation or reduction, you may also hear it called "normalizing" or "executing" an expression.

2.4 Functions

Expressions are the most basic unit of a Haskell program, and functions are a specific type of expression. Functions in Haskell are related to functions in mathematics, which is to say they map an input or set of inputs to an output. A function is an expression that is applied to an argument (or parameter) and always returns a result. Because they are built purely of expressions, they will always evaluate to the same result when given the same values.

As in the lambda calculus, all functions in Haskell take one argument and return one result. The way to think of this is that, in Haskell, when it seems we are passing multiple arguments to a function, we are actually applying a series of nested functions, each to one argument. This is called currying, and it will be addressed in greater detail later.

You may have noticed that the expressions we've looked at so far use literal values with no variables or abstractions. Functions allow us to abstract the parts of code we'd want to reuse for different literal values. Instead of nesting addition expressions, for example, we could write a function that would add the value we wanted wherever we called that function.

For example, say you had a bunch of simple expressions you needed to multiply by 3. You could keep entering them as individual expressions like this:

```
Prelude> (1 + 2) * 3
9
Prelude> (4 + 5) * 3
27
Prelude> (10 + 5) * 3
45
```

But you don't want to do that. Functions are how we factor out the pattern into something we can reuse with different inputs. You do that by naming the function and introducing an independent variable as the argument to the function. Functions can also appear in the expressions that form the bodies of other functions or be used as arguments to functions, just as any other value can be.

In this case, we have a series of expressions that we want to multiply by 3. Let's think in terms of a function: what part is common to all the expressions? What part varies? We know we have to give functions a name and apply them to an argument, so what could we call this function and what sort of argument might we apply it to?

The common pattern is the `* 3` bit. The part that varies is the addition expression before it, so we will make that a variable. We will name our function and apply it to the variable. When we input a value for the variable, our function will evaluate that, multiply it by 3, and return a result. In the next section, we will formalize this into a proper Haskell function.

Defining functions

Function definitions all share a few things in common. First, they start with the name of the function. This is followed by the formal arguments or parameters of the function, separated only by white space. Next there is an equal sign, which, notably, does not imply equality of value. Finally there is an expression that is the body of the function and can be evaluated to return a value.

Defining functions in a normal Haskell source code file and in GHCi are a little different. To introduce definitions of values or functions in GHCi you must use `let`, which looks like this:

```
Prelude> let triple x = x * 3
```

In a source file we would enter it like this:

```
triple x = x * 3
```

Let's examine each part of that:

```
  triple x   =   x * 3
-- [1]  [2] [3]  [ 4 ]
```

1. Name of the function we are defining. Note that it is lowercase.
2. Argument to our function. The arguments to our function correspond to the "head" of a lambda.
3. The = is used to define (or *declare*) values and functions. Reminder: this is *not* how we express equality between two values in Haskell.
4. Body of the function, an expression that could be evaluated if the function is applied to a value. What `triple` is applied to will be the value which the argument x is bound to. Here the expression `x * 3` constitutes the body of the function. So, if you have an expression like `triple 6`, x is bound to 6. Since you've applied the function, you can also replace the fully applied function with its body and bound arguments.

Capitalization matters! Names of modules and names of types, such as `Integer`, start with a capital letter. They can also be `CamelCase` when you want to maintain readability of multiple-word names.

Function names start with lowercase letters. Sometimes for clarity in function names, you may want `camelBack` style, and that is good style provided the very first letter remains lowercase.

Variables are lowercase.

Playing with the triple function First, try entering the `triple` function directly into the REPL using `let`. Now call the function by name and introduce a numeric value for the x argument:

```
Prelude> triple 2
6
```

Next, enter the second version (the one without `let`) into a source file and save the file. Load it into GHCi, using the `:load` or `:l` command. Once it's loaded, you can call the function at the prompt using the function name, `triple`, followed by a numeric value, just as you did in the REPL example above. Try using different values for x — integer values or other arithmetic expressions. Then try changing the function itself in the source file and reloading it to see what changes.

Evaluating functions

Calling the function by name and introducing a value for the x argument makes our function a reducible expression. In a pure functional language like Haskell, we can replace applications of functions with their definitions and get the same result, just like in math. As a result when we see:

```
triple 2
```

We can know that, since triple is defined as `x = x * 3`, the expression is equivalent to:

```
 triple            2
(triple x = x * 3) 2
(triple 2 = 2 * 3)
2 * 3
6
```

What we've done here is apply `triple` to the value 2 and then reduce the expression to the final result 6. Our expression `triple 2` is in canonical or *normal form* when it reaches the number 6 because the value 6 has no remaining reducible expressions.

You may notice that after loading code from a source file, the GHCi prompt is no longer `Prelude>`. To return to the `Prelude>` prompt, use the command `:m`, which is short for `:module`. This will unload the file from GHCi, so the code in that file will no longer be in scope in your REPL.

Conventions for variables

Haskell uses a lot of variables, and some conventions have developed. It's not critical that you memorize this, because for the most part, these are merely conventions, but familiarizing yourself with them will help you read Haskell code.

Type variables (that is, variables in type signatures) generally start at a and go from there: a, b, c, and so forth. You may occasionally see them with numbers appended to them, e.g., $a1$.

Functions can be used as arguments and in that case are typically labeled with variables starting at f (followed by g and so on). They may sometimes have numbers appended (e.g., $f1$) and may also sometimes be decorated with the $'$ character as in f'. This would be pronounced "eff-prime," should you have need to say it aloud. Usually this denotes a function that is closely related to or a helper function to function f. Functions may also be given variable names that are not on this spectrum as a mnemonic. For example, a function that results in a list of prime numbers might be called p, or a function that fetches some text might be called txt. Variables do not have to be a single letter, though they often are.

Arguments to functions are most often given names starting at x, again occasionally seen numbered as in $x1$. Other single-letter variable names may be chosen when they serve a mnemonic role, such as choosing r to represent a value that is the radius of a circle.

If you have a list of things you have named x, by convention that will usually be called xs, that is, the plural of x. You will see this convention often in the form `(x:xs)`, which means you have a list in which the head of the list is x and the rest of the list is xs.

All of these, though, are merely conventions, not definite rules. While we will generally adhere to the conventions in this book, any Haskell code you see out in the wild may not. Calling a type variable x instead of a is not going to break anything. As in the lambda calculus, the names don't have any inherent meaning. We offer this information as a descriptive guide of Haskell conventions, not as rules you must follow in your own code.

Intermission: Exercises

1. Given the following lines of code as they might appear in a source file, how would you change them to use them directly in the REPL?

```
half x = x / 2

square x = x * x
```

2. Write one function that can accept one argument and work for all the following expressions. Be sure to name the function.

```
3.14 * (5 * 5)
3.14 * (10 * 10)
3.14 * (2 * 2)
3.14 * (4 * 4)
```

2.5 Infix operators

Functions in Haskell default to prefix syntax, meaning that the function being applied is at the beginning of the expression rather than the middle. We saw that with our `triple` function, and we see it with standard functions such as the `identity` or `id` function. This function just returns whatever value it is given as an argument:

```
Prelude> id 1
1
```

While this is the default syntax for functions, not all functions are prefix. There are a group of operators, such as the arithmetic operators we've been using, that are indeed functions (they apply to arguments to produce an output) but appear by default in an infix position.

Operators are functions which can be used in infix style. All operators are functions; not all functions are operators. While `triple` and `id` are prefix functions (*not* operators), the `+` function is an infix operator:

```
Prelude> 1 + 1
2
```

Now we'll try a few other mathematical operators:

```
Prelude> 100 + 100
200
Prelude> 768395 * 21356345
16410108716275
Prelude> 123123 / 123
1001.0
Prelude> 476 - 36
440
Prelude> 10 / 4
2.5
```

You can sometimes use functions in an infix or prefix style, with a small change in syntax:

```
Prelude> 10 `div` 4
2
Prelude> div 10 4
2
```

Associativity and precedence

As you may remember from your math classes, there's a default associativity and precedence to the infix operators `(*)`, `(+)`, `(-)`, and `(/)`.

We can ask GHCi for information such as associativity and precedence of operators and functions by using the `:info` command. When you ask GHCi for the `:info` about an operator or function, it provides the type information and tells you whether it's an infix operator with precedence. We will focus on the precedence and associativity for infix operators. Here's what the code in Prelude says for `(*)`, `(+)`, and `(-)` at time of writing:

```
:info (*)
infixl  7   *
-- [1] [2] [3]

:info (+) (-)
infixl 6  +, -
--        [4]
```

1. `infixl` means infix operator, left associative
2. 7 is the precedence: higher is applied first, on a scale of 0-9.
3. Infix function name: in this case, multiplication
4. We use the comma here to assign left-associativity and precedence 6 for two functions `(+)` and `(-)`

Here `infixl` means the operator associates to the left. Let's play with parentheses and see what this means. Continue to follow along with the code via the REPL:

```
-- this
2 * 3 * 4

-- is evaluated as if it was
(2 * 3) * 4
-- Because of left-associativity from infixl
```

Here's an example of a right-associative infix operator:

```
Prelude> :info (^)
infixr  8   ^
-- [1] [2] [3]
```

1. `infixr` means infix operator, right associative

2. 8 is the precedence. Higher precedence, indicated by higher numbers, is applied first, so this is higher precedence than multiplication (7), addition, or subtraction (both 6).
3. Infix function name: in this case, exponentiation.

It was hard to tell before why associativity mattered with multiplication, because multiplication is associative. So shifting the parentheses around never changes the result. Exponentiation, however, is not associative and thus makes a prime candidate for demonstrating left vs. right associativity.

```
Prelude> 2 ^ 3 ^ 4
2417851639229258349412352
Prelude> 2 ^ (3 ^ 4)
2417851639229258349412352
Prelude> (2 ^ 3) ^ 4
4096
```

As you can see, adding parentheses starting from the right-hand side of the expression when the operator is right-associative doesn't change anything. However, if we parenthesize from the *left*, we get a different result when the expression is evaluated.

Your intuitions about precedence, associativity, and parenthesization from math classes will generally hold in Haskell:

```
2 + 3 * 4
```

```
(2 + 3) * 4
```

What's the difference between these two? Why are they different?

Intermission: Exercises

Below are some pairs of functions that are alike except for parenthesization. Read them carefully and decide if the parentheses change the results of the function. Check your work in GHCi.

1. a) `8 + 7 * 9`
   b) `(8 + 7) * 9`
2. a) `perimeter x y = (x * 2) + (y * 2)`
   b) `perimeter x y = x * 2 + y * 2`
3. a) `f x = x / 2 + 9`
   b) `f x = x / (2 + 9)`

2.6 Declaring values

The order of declarations in a source code file doesn't matter because GHCi loads the entire file at once, so it knows all the values that have been specified, no matter what order they appear in. On the other hand, when you enter them one-by-one into the REPL, the order does matter.

For example, we can declare a series of expressions in the REPL like this:

```
Prelude> let y = 10
Prelude> let x = 10 * 5 + y
Prelude> let myResult = x * 5
```

As we've seen above when we worked with the `triple` function, we have to use `let` to declare something in the REPL.

We can now just type the names of the values and hit enter to see their values:

```
Prelude> x
60
Prelude> y
10
Prelude> myResult
300
```

To declare the same values in a file, such as `learn.hs`, we write the following:

```
-- learn.hs

module Learn where
-- First, we declare the name of our module so
-- it can be imported by name in a project.
-- We won't be doing a project of this size
-- for a while yet.

x = 10 * 5 + y

myResult = x * 5

y = 10
```

Remember module names are capitalized, unlike function names. Also, in this function name, we've used camelBack case: the first letter is still lowercase, but we use an uppercase to delineate a word boundary for readability.

Troubleshooting

It is easy to make mistakes in the process of typing `learn.hs` into your editor. We'll look at a few common mistakes in this section. One thing to keep in mind is that indentation of Haskell code is significant and can change the meaning of the code. Incorrect indentation of code can also break your code. Reminder: use spaces *not* tabs to indent your source code.

In general, whitespace is significant in Haskell. Efficient use of whitespace makes the syntax more concise. This can take some getting used to if you've been working in another programming language. Whitespace is often the only mark of a function call, unless parentheses are necessary due to conflicting precedence. Trailing whitespace, that is, extraneous whitespaces at the end of lines of code, is considered bad style.

In source code files, indentation is important and often replaces syntactic markers like curly brackets, semicolons, and parentheses. The basic rule is that code that is part of an expression should be indented under the beginning of that expression, even when the beginning of the expression is not at the leftmost margin. Furthermore, parts of the expression that are grouped should be indented to the same level. For example, in a block of code introduced by `let` or `do`, you might see something like this:

```
let
  x = 3
  y = 4

-- or

let x = 3
    y = 4

-- Note that this code won't work directly in a
-- source file without embedding in a
-- top-level declaration
```

Notice that the two definitions that are part of the expression line up in either case. It is incorrect to write:

```
let x = 3
 y = 4

-- or

let
 x = 3
  y = 4
```

If you have an expression that has multiple parts, your indentation will follow a pattern like this:

```
foo x =
    let y = x * 2
        z = x ^ 2
    in 2 * y * z
```

Notice that the definitions of y and z line up, and the definitions of `let` and `in` are also aligned. As you work through the book, try to pay careful attention to the indentation patterns as we have them printed. There are many cases where improper indentation will actually cause code not to work.

Also, when you write Haskell code, we reiterate here that you want to use spaces and *not* tabs for indentation. Using spaces will save you a nontrivial amount of grief. If you don't know how to make your editor only use spaces for indentation, *look it up*!

If you make a mistake like breaking up the declaration of x such that the rest of the expression began at the beginning of the next line:

```
module Learn where
-- First declare the name of our module so it
-- can be imported by name in a project.
-- We won't do this for a while yet.

x = 10
* 5 + y

myResult = x * 5

y = 10
```

You might see an error like:

```
Prelude> :l code/learn.hs
[1 of 1] Compiling Learn

code/learn.hs:10:1: parse error on input ‘*’
Failed, modules loaded: none.
```

Note that the first line of the error message tells you where the error occurred: `code/learn.hs:10:1` indicates that the mistake is in line 10, column 1, of the named file. That can make it easier to find the problem that needs to be fixed. Please note that the exact line and column numbers in your own error messages might be different from ours, depending on how you've entered the code into the file.

The way to fix this is to either put it all on one line, like this:

```
x = 10 * 5 + y
```

or to make certain that when you break up lines of code that the second line begins at least one space from the beginning of that line (either of the following should work):

```
x = 10
 * 5 + y

-- or

x = 10
    * 5 + y
```

The second one looks a little better, and generally you should reserve breaking up of lines for when you have code exceeding 100 columns in width.

Another possible error is not starting a declaration at the beginning (left) column of the line:

```
-- learn.hs

module Learn where

 x = 10 * 5 + y

myResult = x * 5

y = 10
```

See that space before x? That will cause an error like:

```
Prelude> :l code/learn.hs
[1 of 1] Compiling Learn

code/learn.hs:11:1: parse error on input ‘myResult’
Failed, modules loaded: none.
```

This may confuse you, as `myResult` is not where you need to modify your code. The error is only an extraneous space, but all declarations in the module must start at the same column. The column that all declarations within a module must start in is determined by the first declaration in the module. In this case, the error message gives a location that is different from where you should fix the problem because all the compiler knows is that the declaration of x made a single space the appropriate indentation for all declarations within that module, and the declaration of `myResult` began a column too early.

It is possible to fix this error by indenting the `myResult` and y declarations to the same level as the indented x declaration:

```
-- learn.hs

module Learn where

 x = 10 * 5 + y

 myResult = x * 5

 y = 10
```

However, this is considered bad style and is not standard Haskell practice. There is almost never a good reason to indent all your declarations in this way, but noting this gives us some idea of how the compiler is reading the code. It is better, when confronted with an error message like this, to make sure that your first declaration is at the leftmost margin and proceed from there.

Another possible mistake is that you might've missed the second `-` in the `--` used to comment out source lines of code.

So this code:

```
- learn.hs

module Learn where
-- First declare the name of our module so it
-- can be imported by name in a project.
-- We won't do this for a while yet.

x = 10 * 5 + y

myResult = x * 5

y = 10
```

will cause this error:

```
Prelude> :r
[1 of 1] Compiling Main

code/learn.hs:7:1: parse error on input ‘module’
Failed, modules loaded: none.
```

Note again that it says the parse error occurred at the beginning of the module declaration, but the issue is actually that `- learn.hs` had only one `-` when it needed two to form a syntactically correct Haskell comment.

Now we can see how to work with code that is saved in a source file from GHCi without manually copying and pasting the definitions into our REPL. Assuming we open our REPL in the same directory as we have `learn.hs` saved, we can do the following:

```
Prelude> :load learn.hs
[1 of 1] Compiling Learn
```

```
Ok, modules loaded: Learn.
Prelude> x
60
Prelude> y
10
Prelude> myResult
300
```

Intermission: Exercises

The following code samples are broken and won't compile. The first two are as you might enter into the REPL; the third is from a source file. Find the mistakes and fix them so that they will.

1. `let area x = 3. 14 * (x * x)`
2. `let double x = b * 2`
3. ```
   x = 7
    y = 10
   f = x + y
   ```

2.7 Arithmetic functions in Haskell

Let's work with a wider variety of arithmetic functions in Haskell. We will use some common operators and functions for arithmetic. The infix operators, their names, and their use in Haskell are listed here:
   ```

Operator	Name	Purpose/application
+	plus	addition
-	minus	substraction
*	asterisk	multiplication
/	slash	fractional division
div	divide	integral division, round down
mod	modulo	remainder after division
quot	quotient	integral division, round towards zero
rem	remainder	remainder after division

At the risk of stating the obvious, "integral" division refers to division of integers. Because it's integral and not fractional, it takes integers as arguments and returns integers as results. That's why the results are rounded.

Here's an example of each in the REPL:

```
Prelude> 1 + 1
2
Prelude> 1 - 1
0
Prelude> 1 * 1
1
Prelude> 1 / 1
1.0
Prelude> div 1 1
1
Prelude> mod 1 1
0
Prelude> quot 1 1
1
Prelude> rem 1 1
0
```

Always use `div` for integral division unless you know what you're doing. However, if the remainder of a division is the value you need, `rem` is usually the function you want. We will look at `/` in more detail later, as that will require some explanation of types and typeclasses.

2.8 Negative numbers

Due to the interaction of parentheses, currying, and infix syntax, negative numbers get special treatment in Haskell.

If you want a value that is a negative number by itself, this will work just fine:

```
Prelude> -1000
-1000
```

However, this will not work in some cases:

```
Prelude> 1000 + -9
<interactive>:3:1:
    Precedence parsing error
        cannot mix ‘+’ [infixl 6] and
        prefix `-` [infixl 6]
           in the same infix expression
```

Fortunately, we were told about our mistake before any of our code was executed. Note how the error message tells you the problem has to do with precedence. Addition and subtraction have the same precedence (6), and GHCi doesn't know how to resolve the precedence and evaluate the expression. We need to make a small change before we can add a positive and a negative number together:

```
Prelude> 1000 + (-9)
991
```

The negation of numbers in Haskell by the use of a unary `-` is a form of *syntactic sugar*. Syntax is the grammar and structure of the text we use to express programs, and syntactic sugar is a means for us to make that text easier to read and write. Syntactic sugar is so-called because while it can make the typing or reading of the code nicer, it changes nothing about the

semantics of our programs and doesn't change how we solve our problems in code. Typically when code with syntactic sugar is processed by our REPL or compiler, a simple transformation from the shorter ("sweeter") form to a more verbose, truer representation is performed after the code has been parsed.

In the specific case of `-`, the syntax sugar means the operator now has two possible interpretations. The two possible interpretations of the syntactic `-` are that `-` is being used as an alias for `negate` or that it is the subtraction function. The following are semantically identical (that is, they have the same meaning, despite different syntax) because the `-` is translated into `negate`:

```
Prelude> 2000 + (-1234)
766

Prelude> 2000 + (negate 1234)
766
```

Whereas this is a case of `-` being used for subtraction:

```
Prelude> 2000 - 1234
766
```

Fortunately, syntactic overloading like this isn't common in Haskell.

2.9 Parenthesizing infix functions

There are times when you want to refer to an infix function without applying any arguments, and there are also times when you want to use them as prefix operators instead of infix. In both cases you must wrap the operator in parentheses. We will see more examples of the former case later in the book. For now, let's look at how we use infix operators as prefixes.

If your infix function is `>>` then you must write `(>>)` to refer to it as a value. `(+)` is the addition infix function without any arguments applied yet and

`(+1)` is the same addition function but with one argument applied, making it return the next argument it's applied to plus one:

```
Prelude> 1 + 2
3
Prelude> (+) 1 2
3
Prelude> (+1) 2
3
```

The last case is known as *sectioning* and allows you to pass around partially-applied functions. With commutative functions, such as addition, it makes no difference if you use `(+1)` or `(1+)` because the order of the arguments won't change the result.

If you use sectioning with a function that is not commutative, the order matters:

```
Prelude> (1/) 2
0.5
Prelude> (/1) 2
2.0
```

Subtraction, `(-)`, is a special case. These will work:

```
Prelude> 2 - 1
1
Prelude> (-) 2 1
1
```

The following, however, won't work:

```
Prelude> (-2) 1
```

Enclosing a value inside the parentheses with the `-` indicates to GHCi that it's the argument of a function. Because the `-` function represents negation,

not subtraction, when it's applied to a single argument, GHCi does not know what to do with that, and so it returns an error message. Here, `-` is a case of syntactic overloading disambiguated by how it is used.

You can use sectioning for subtraction, but it must be the first argument:

```
Prelude> let x = 5
Prelude> let y = (1 -)
Prelude> y x
-4
```

It may not be immediately obvious why you would ever want to do this, but you will see this syntax used throughout the book, particularly once we start wanting to apply functions to each value inside a list.

2.10 Laws for quotients and remainders

Programming often makes use of more division and remainder functions than standard arithmetic does, and it's helpful to be familiar with the laws about quot/rem and div/mod.[1]We'll take a look at those here.

```
(quot x y)*y + (rem x y) == x

(div x y)*y + (mod x y) == x
```

We won't walk through a proof exercise, but we can demonstrate these laws a bit:

```
(quot x y)*y + (rem x y)

Given x is 10 and y is (-4)

(quot 10 (-4))*(-4) + (rem 10 (-4))
```

[1]From Lennart Augustsson's blog `http://augustss.blogspot.com/`

```
quot 10 (-4) == (-2)  and  rem 10 (-4) == 2

(-2)*(-4) + (2) == 10

10 == x == yeppers.
```

Now for div/mod:

```
(div x y)*y + (mod x y)

Given x is 10 and y is (-4)

(div 10 (-4))*(-4) + (mod 10 (-4))

div 10 (-4) == (-3)  and  mod 10 (-4) == -2

(-3)*(-4) + (-2) == 10

10 == x == yeppers.
```

2.11 Evaluation

When we talk about reducing an expression, we're talking about evaluating the terms until it reaches its simplest form. Once a term has reached its simplest form, we say that it is irreducible or finished evaluating. Usually, we call this a value. Haskell uses a non-strict evaluation (sometimes called "lazy evaluation") strategy which defers evaluation of terms until they're forced by other terms referring to them. We will return to this concept several times throughout the book as it takes time to fully understand.

Values are irreducible, but applications of functions to arguments are reducible. Reducing an expression means evaluating the terms until you're left with an irreducible value. As in the lambda calculus, application is evaluation, another theme that we will return to throughout the book.

Values are expressions, but cannot be reduced further. Values are a terminal point of reduction:

```
1
"Icarus"
```

The following expressions can be reduced (evaluated, if you will) to a value:

```
1 + 1
2 * 3 + 1
```

Each can be evaluated in the REPL, which reduces the expressions and then prints what it reduced to.

2.12 Let and where

We can use `let` and `where` to introduce names for expressions.

We'll start with an example of `where`:

```
-- FunctionWithWhere.hs
module FunctionWithWhere where

printInc n = print plusTwo
  where plusTwo = n + 2
```

And if we use this in the REPL:

```
Prelude> :l FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere ...
Ok, modules loaded: FunctionWithWhere.
Prelude> printInc 1
3
Prelude>
```

Now we have the same function, but using `let` in the place of where:

```
-- FunctionWithLet.hs
module FunctionWithLet where

printInc2 n = let plusTwo = n + 2
              in print plusTwo
```

When you see `let` followed by `in` you're looking at a *let expression*. Here's that function in the REPL:

```
Prelude> :load FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet ...
Ok, modules loaded: FunctionWithLet.
Prelude> printInc2 3
5
```

If you loaded the `FunctionWithLet` version of `printInc2` in the same REPL session as `FunctionWithWhere` then it will have unloaded the old version before loading the new one. That is one limitation of the `:load` command in GHCi. As we build larger projects that require having multiple modules in scope, we will use a project manager called Cabal and a `cabal repl` rather than GHCi itself.

Here's a demonstration of what is meant by GHCi unloading everything that had been defined in the REPL when you `:load` a file:

```
Prelude> :load FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere ...
Ok, modules loaded: FunctionWithWhere.
Prelude> print
print     printInc
Prelude> printInc 1
3
Prelude> :load FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet ...
```

```
Ok, modules loaded: FunctionWithLet.
Prelude> printInc2 10
12
Prelude> printInc 10

<interactive>:6:1:
    Not in scope: ‘printInc’
    Perhaps you meant ‘printInc2’ (line 4)
```

`printInc` isn't in scope anymore because GHCi unloaded everything you'd defined or loaded after you used `:load` to load the `FunctionWithLet.hs` source file. Scope is the area of source code where a binding of a variable applies.

Intermission: Exercises

Now for some exercises. First, determine in your head what the following expressions will return, then validate in the REPL:

1. `let x = 5 in x`
2. `let x = 5 in x * x`
3. `let x = 5; y = 6 in x * y`
4. `let x = 3; y = 1000 in x + 3`

Above, you entered some `let` expressions into your REPL to evaluate them. Now, we're going to open a file and rewrite some `let` expressions into `where` clauses. You will have to give the value you're binding a name, although the name can be just a letter if you like. For example,

```
-- this should work in GHCi
let x = 5; y = 6 in x * y
```

could be rewritten as

```
-- put this in a file
mult1      = x * y
   where x = 5
         y = 6
```

Making the equals signs line up is a stylistic choice. As long as the expressions are nested in that way, the equals signs do not have to line up. But notice we use a name that we will use to refer to this value in the REPL:

```
*Main> :l practice.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> mult1
30
```

Note: the filename you choose is unimportant except for the .hs extension.

The lambdas beneath let expressions

It turns out that there's a fairly straightforward way to express lambdas in Haskell with the provided anonymous function syntax. Recall the identity function that we saw in the lambda calculus chapter:

```
λ x . x
```

Anonymous function syntax in Haskell uses a backslash to represent a lambda. It looks quite similar to the way it looks in the lambda calculus, and we can use it by wrapping it in parentheses and applying it to values:

```
Prelude> (\x -> x) 0
0
Prelude> (\x -> x) 1
1
```

```
Prelude> (\x -> x) "blah"
"blah"
```

We can also define the same function in the REPL using `let`:

```
Prelude> let id = \x -> x
Prelude> id 0
0
Prelude> id 1
1
```

Or we can define it this way:

```
Prelude> let id x = x
Prelude> id 0
0
Prelude> id 1
1
```

`let` expressions happen to be sugar for lambdas. In fact, pretty much all of a pure functional programming language turns out to be human-friendly gift-wrapping for lambdas. Let us translate a few `let` expressions into their lambda forms!

```
let a = b in c

-- equivalent to

(\a -> c) b

-- or a little less abstractly

let x = 10 in x + 9001

(\x -> x + 9001) 10
```

We can do a similar translation with `where`, although there are minute differences that we will not address here:

```
c where a = b

-- equivalent to

(\a -> c) b

-- Something more concrete again

x + 9001 where x = 10

(\x -> x + 9001) 10
```

We won't break down every single Haskell construct into the underlying lambdas, but try to keep this model in mind as you proceed.

More exercises!

Rewrite the following `let` expressions into declarations with `where` clauses:

1. `let x = 3; y = 1000 in x * 3 + y`
2. `let y = 10; x = 10 * 5 + y in x * 5`
3. `let x = 7; y = negate x; z = y * 10 in z / x + y`

2.13 Chapter Exercises

The goal for all the following exercises is just to get you playing with code and forming hypotheses about what it should do. Read the code carefully, using what we've learned so far. Generate a hypothesis about what you think the code will do. Play with it in the REPL and find out where you were right or wrong.

Parenthesization

Here we've listed the information that GHCi gives us for various infix operators. We have left the type signatures in this time, although it is not directly relevant to the following exercises. This will give you a chance to look at the types if you're curious and also provide a more accurate picture of the `:info` command.

```
Prelude> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a
infixr 8 ^

Prelude> :info (*)
class Num a where
  (*) :: a -> a -> a
infixl 7 *

Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
infixl 6 +

Prelude> :info (-)
class Num a where
  (-) :: a -> a -> a
infixl 6 -

Prelude> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $
```

We should take a moment to explain and demonstrate the `($)` operator as you will run into it fairly frequently in Haskell code. The good news is it does almost nothing. The bad news is this fact sometimes trips people up.

First, here's the definition of `($)`:

```
f $ a = f a
```

Immediately this seems a bit pointless until we remember that it's defined as an infix operator with the lowest possible precedence. The ($) operator is a convenience for when you want to express something with fewer pairs of parentheses.

Example of ($) in some expressions:

```
Prelude> (2^) $ 2 + 2
16
Prelude> (2^) (2 + 2)
16
Prelude> (2^) 2 + 2
6
```

If you like, a way to understand ($) in words is: "evaluate everything to the right of me first."

Also note that you can stack up multiple uses of ($) in the same expression. For example, this works:

```
Prelude> (2^) $ (+2) $ 3*2
256
```

But this does not:

```
Prelude> (2^) $ 2 + 2 $ (*30)
-- A rather long and ugly type error about trying to
-- use numbers as if they were functions follows.
```

We can see for ourselves why this code doesn't make sense if we examine the reduction steps.

```
-- Remember ($)'s definition
f $ a = f a

(2^) $ 2 + 2 $ (*30)
-- Given the right-associativity (infixr) of $
-- we must begin at the right-most position.
2 + 2 $ (*30)
-- reduce ($)
(2 + 2) (*30)
-- then we must evaluate (2 + 2) before we can apply it
4 (*30)
-- This doesn't make sense, we can't apply 4
-- as if it was a function to the argument (*30)!
```

Now let's flip that expression around a bit so it works and then walk through a reduction:

```
(2^) $ (*30) $ 2 + 2
-- must evaluate right-side first
(2^) $ (*30) $ 2 + 2
-- application of the function (*30) to the
-- expression (2 + 2) forces evaluation
(2^) $ (*30) 4
-- then we reduce (*30) 4
(2^) $ 120
-- reduce ($) again.
(2^) 120
-- reduce (2^)
1329227995784915872903807060280344576
```

Given what we know about the precedence of `(*)`, `(+)`, and `(^)`, how can we parenthesize the following expressions more explicitly without changing their results? Put together an answer you think is correct, then test in the GHCi REPL.

Example:

```
-- We want to make this more explicit
2 + 2 * 3 - 3

-- this will produce the same result
2 + (2 * 3) - 3
```

Attempt the above on the following expressions.

1. `2 + 2 * 3 - 1`
2. `(^) 10 $ 1 + 1`
3. `2 ^ 2 * 4 ^ 5 + 1`

Equivalent expressions

Which of the following pairs of expressions will return the same result when evaluated? Try to reason them out in your head by reading the code and then enter them into the REPL to check your work:

1. `1 + 1`

   `2`

2. `10 ^ 2`

   `10 + 9 * 10`

3. `400 - 37`

   `(-) 37 400`

4. `100 `div` 3`

   `100 / 3`

5. `2 * 5 + 18`

   `2 * (5 + 18)`

More fun with functions

Here is a bit of code as it might be entered into a source file. Remember that when you write code in a source file, the order is unimportant, but when writing code directly into the REPL the order does matter. Given that, look at this code and rewrite it such that it could be evaluated in the REPL (remember: you'll need `let` when entering it directly into the REPL). Be sure to enter your code into the REPL to make sure it evaluates correctly.

```
z = 7

x = y ^ 2

waxOn = x * 5

y = z + 8
```

1. Now you have a value called `waxOn` in your REPL. What do you think will happen if you enter:

   ```
   10 + waxOn
   -- or
   (+10) waxOn
   -- or
   (-) 15 waxOn
   -- or
   (-) waxOn 15
   ```

2. Earlier we looked at a function called `triple`. While your REPL has `waxOn` in session, re-enter the `triple` function at the prompt:

   ```
   let triple x = x * 3
   ```

3. Now, what will happen if we enter this at our GHCi prompt. Try to reason out what you think will happen first, considering what role `waxOn` is playing in this function call. Then enter it, see what does happen, and check your understanding:

   ```
   triple waxOn
   ```

4. Rewrite `waxOn` as a function with a `where` clause in your source file. Load it into your REPL and make sure it still works as expected!

5. Now to the same source file where you have `waxOn`, add the `triple` function. Remember: You don't need `let` and the function name should be at the left margin (that is, not nested as one of the `waxOn` expressions). Make sure it works by loading it into your REPL and then entering `triple waxOn` again at the REPL prompt. You should have the same answer as you did above.

6. Now, without changing what you've done so far in that file, add a new function called `waxOff` that looks like this:

   ```
   waxOff x = triple x
   ```

7. Load the source file into your REPL and enter `waxOff waxOn` at the prompt.

   You now have a function, `waxOff` that can be applied to a variety of arguments — not just `waxOn` but any (numeric) value you want to put in for x. Play with that a bit. What is the result of `waxOff 10` or `waxOff (-50)`? Try modifying your `waxOff` function to do something new — perhaps you want to first triple the x value and then square it or divide it by 10. Just spend some time getting comfortable with modifying the source file code, reloading it, and checking your modification in the REPL.

2.14 Definitions

1. An *argument* (also, parameter) is an input to a function. Where we have the function `f x = x + 2` which takes an argument and returns that value added to 2, x is the argument or parameter to our function.

2. An *expression* is a combination of symbols that conforms to syntactic rules and can be evaluated to some result. In Haskell, an expression is a well-structured combination of constants, variables, and functions. While irreducible constants are technically expressions, we usually refer to those as "values", so we usually mean "reducible expression" when we use this term.

3. A *redex* is a reducible expression.

4. A *value* is an expression that cannot be reduced or evaluated any further. `2 * 2` is an expression, but not a value, whereas what it evaluates to, 4, is a value.

5. A *function* is a mathematical object whose capabilities are limited to being applied to an argument and returning a result. Functions can be described as a list of ordered pairs of their inputs and the resulting outputs, like a mapping. Given the function `f x = x + 2` applied to the argument 2, we would have the ordered pair `(2, 4)` of its input and output.

6. *Infix notation* is the style used in arithmetic and logic. Infix means that the operator is placed between the operands or *arguments*. An example would be the plus sign in an expression like `2 + 2`.

7. *Operators* are functions that are infix by default. In Haskell, operators must use symbols and not alphanumeric characters.

8. *Syntactic sugar* is syntax within a programming language designed to make expressions easier to write or read.

2.15 Follow-up resources

1. Haskell wiki article on Let vs. Where
   `https://wiki.haskell.org/Let_vs._Where`
2. Gabriel Gonzalez; How to desugar Haskell code
   `http://www.haskellforall.com/2014/10/how-to-desugar-haskell-code.html`

Chapter 3

Strings

Simple operations with text

> Like punning, programming is a play on words
>
> Alan Perlis

3.1 Printing strings

So far we've been looking at doing arithmetic using simple expressions. In this section, we will look at another type of basic expression that processes a different type of data called `String`.

Most programming languages refer to the data structures used to contain text as "strings," usually represented as sequences, or lists, of characters. In this section, we will

- take an introductory look at types to understand the data structure called String;
- talk about the special syntax, or syntactic sugar, used for strings;
- print strings in the REPL environment;
- work with some simple functions that operate on this datatype.

3.2 A first look at types

First, since we will be working with strings, we want to start by understanding what these data structures are in Haskell and a bit of special syntax we use for them. We haven't talked much about types yet, although you saw some examples of them in the last chapter. Types are important in Haskell, and the next two chapters are entirely devoted to them.

Types are a way of categorizing values. There are several types for numbers, for example, depending on whether they are integers, fractional numbers, etc. There is a type for boolean values, specifically the values `True` and `False`. The types we are primarily concerned with in this chapter are `Char` 'character' and `String`. `String`s are lists of characters.

It is easy to find out the type of a value, expression, or function in GHCi. We do this with the `:type` command.

Open up your REPL, enter `:type 'a'` at the prompt, and you should see something like this:

```
Prelude> :type 'a'
'a' :: Char
```

We need to highlight a few things here. First, we've enclosed our character in single quotes. This lets GHCi know that the character is not a variable. If you enter `:type a` instead, it will think it's a variable and give you an error message that the *a* is not in scope. That is, the variable *a* hasn't been defined (is not in scope), so it has no way to know what the type of it is.

Second, the `::` symbol is read as "has the type." You'll see this often in Haskell. Whenever you see that double colon, you know you're looking at a type signature. A type signature is a line of code that defines the types for a value, expression, or function.

And, finally, there is `Char`, the type. Char is the type that includes alphabetic characters, unicode characters, symbols, etc. So, asking GHCi `:type 'a'`, that is, "what is the type of 'a'?", gives us the information, `'a' :: Char`, that is, "'a' has the type of Char."

Now, let's try a string of text. This time we have to use double quotation marks, not single, to tell GHCi we have a string, not a single character:

```
Prelude> :type "Hello!"
"Hello!" :: [Char]
```

We have something new in the type information. The square brackets around Char here are the syntactic sugar for a list. When we talk about lists in more detail later, we'll see why this is considered syntactic sugar; for now, we just need to understand that GHCi says "Hello!" has the type *list of Char*.

3.3 Printing simple strings

Now, let's look at some simple commands for printing strings of text in the REPL:

```
Prelude> print "hello world!"
"hello world!"
```

Here we've used the command `print` to tell GHCi to print the string to the display, so it does, with the quotation marks still around it.

Other commands we can use to tell GHCi to print strings of text into the display have slightly different results:

```
Prelude> putStrLn "hello world!"
hello world!
Prelude>

Prelude> putStr "hello world!"
hello world!Prelude>
```

You can probably see that `putStr` and `putStrLn` are similar to each other, with one key difference. We also notice that both of these commands print the string to the display without the quotation marks. This is because, while they are superficially similar to `print`, they actually have a different type than `print` does. Functions that are similar on the surface can behave differently depending on the type or category they belong to.

Next, let's take a look at how to do these things from source files. Type the following into a file named `print1.hs`:

```
-- print1.hs
module Print1 where

main :: IO ()
main = putStrLn "hello world!"
```

Here's what you should see when you run this code by loading it in GHCi and running `main`:

```
Prelude> :l print1.hs
```

```
[1 of 1] Compiling Print1
Ok, modules loaded: Print1.
Prelude> main
hello world!
Prelude>
```

The `main` function is the default function when you build an executable or run it in a REPL. As you can see, `main` has the type `IO ()`. `IO` — that's the letters I as in eye and O as in oh — stands for input/output but has a specialized meaning in Haskell. It is a special type used by Haskell when the result of running the program involves side-effects as opposed to being a pure function or expression. Printing to the screen is a side-effect, so printing the output of a module must be wrapped in this `IO` type. When you enter functions directly into the REPL, GHCi implicitly understands and implements `IO` without you having to specify that. Since the `main` function is the default executable function, this bit of code will be in a lot of source files that we build from here on out. We will explain its meaning in more detail in a later chapter.

Let's start another file:

```
-- print2.hs
module Print2 where

main :: IO ()
main = do
  putStrLn "Count to four for me:"
  putStr   "one, two"
  putStr   ", three, and"
  putStrLn " four!"
```

And here's what you should see when you run this one:

```
Prelude> :l print2.hs
[1 of 1] Compiling Print2
Ok, modules loaded: Print2.
Prelude> main
```

```
Count to four for me:
one, two, three, and four!
Prelude>
```

For a bit of fun, change the invocations of `putStr` to `putStrLn` and vice versa. Rerun the program and see what happens.

You'll note the `putStrLn` function prints to the current line, then starts a new line, where `putStr` prints to the current line but doesn't start a new one. The `Ln` in `putStrLn` indicates that it starts a new line.

String concatenation

Concatenation, or to *concatenate* something, means to "link together." Usually when we talk about concatenation in programming we're talking about linear sequences such as lists or strings of text. If we concatenate two strings `"Curry"` and `" Rocks!"` we will get the string `"Curry Rocks!"`. Note the space at the beginning of `" Rocks!"`. Without that, we'd get `"CurryRocks!"`.

Let's start a new text file and type the following:

```haskell
-- print3.hs
module Print3 where

myGreeting :: String
-- The above line reads as: "myGreeting has the type String"
myGreeting = "hello" ++ " world!"
-- Could also be: "hello" ++ " " ++ "world!"
-- to obtain the same result.

hello :: String
hello = "hello"

world :: String
world = "world!"

main :: IO ()
main = do
  putStrLn myGreeting
  putStrLn secondGreeting
  where secondGreeting = concat [hello, " ", world]
```

If you execute this, you should see something like:

```
Prelude> :load print3.hs
[1 of 1] Compiling Print3
Ok, modules loaded: Print3.
*Print3> main
hello world!
hello world!
*Print3>
```

Remember you can use `:m` to return to Prelude if desired.

This little exercise demonstrates a few things:

1. We define values at the top level of a module: (`myGreeting`, `hello`, `world`, and `main`). That is, they are declared globally so that their

values are available to all functions in the module.

2. We specify explicit types for top-level definitions.
3. We concatenate strings with `(++)` and `concat`.

Global versus local definitions

What does it mean for something to be at the top level of a module? It means it is defined globally, not locally. To be locally defined would mean the declaration is nested within some other expression and is not visible to code importing the module. We practiced this in the previous chapter with `let` and `where`. Here's an example for review:

```
module GlobalLocal where

topLevelFunction :: Integer -> Integer
topLevelFunction x = x + woot + topLevelValue
  where woot :: Integer
        woot = 10

topLevelValue :: Integer
topLevelValue = 5
```

In the above, you could import and use `topLevelFunction` or `topLevelValue` from another module. However, `woot` is effectively invisible outside of `topLevelFunction`. The `where` and `let` clauses in Haskell introduce local bindings or declarations. To bind or declare something means to give an expression a name. You could pass around and use an anonymous version of `topLevelFunction` manually, but giving it a name and reusing it by that name is more pleasant and less repetitious. Also note we explicitly declared the type of `woot` in the `where` clause. This wasn't necessary (Haskell's type inference would've figured it out fine), but it was done here to show you how in case you need to. Be sure to load and run this code in your REPL:

```
Prelude> :l Global.hs
```

```
[1 of 1] Compiling GlobalLocal
Ok, modules loaded: GlobalLocal.
*GlobalLocal> topLevelFunction 5
20
```

Experiment with different arguments and make sure you understand the results you're getting by walking through the arithmetic in your head (or on paper).

Intermission: Exercises

1. These lines of code are from a REPL session. Is y in scope for z?

   ```
   Prelude> let x = 5
   Prelude> let y = 7
   Prelude> let z = x * y
   ```

2. These lines of code are from a REPL session. Is h in scope for function g?

   ```
   Prelude> let f = 3
   Prelude> let g = 6 * f + h
   ```

3. This code sample is from a source file. Is everything we need to execute `area` in scope?

   ```
   area d = pi * (r * r)
   r = d / 2
   ```

4. This code is also from a source file. Now are r and d in scope for `area`?

   ```
   area d = pi * (r * r)
     where r = d / 2
   ```

3.4 Type signatures of concatenation functions

Let's look at the types of `(++)` and `concat`. The `++` function is an infix operator. When we need to refer to an infix operator in a position that is not infix — such as when we are using it in a prefix position or having it stand alone in order to query its type — we must put parentheses around it. On the other hand, `concat` is a normal (not infix) function, so parentheses aren't necessary:

```
++     has the type [a] -> [a] -> [a]
concat has the type [[a]] -> [a]

-- Here's how we query that in ghci:
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> :t concat
concat :: [[a]] -> [a]
```

(n.b., Assuming you are using GHC 7.10, if you check this type signature in your REPL, you will find an unexpected and possibly confusing result. We will explain the reason for it later in the book. For your purposes at this point, please understand `Foldable t => t [a]` as being `[[a]]`. The `Foldable t`, for our current purposes, can be thought of as another list. In truth, list is only one of the possible types here — types that have instances of the Foldable typeclass — but right now, lists are the only one we care about.)

But what do these types mean? Here's how we can break it down:

```
(++) :: [a] -> [a] -> [a]
--      [1]    [2]    [3]
```

Everything after the `::` is about our types, not our values. The 'a' inside the list type constructor `[]` is a type variable.

1. Take an argument of type `[a]`, which is a list of elements whose type we don't yet know.
2. Take another argument of type `[a]`, a list of elements whose type we don't know. Because the variables are the same, they must be the same type throughout (a == a).
3. Return a result of type `[a]`

As we'll see, because String is a type of list, the operators we use with strings can also be used on lists of other types, such as lists of numbers. The type `[a]` means that we have a list with elements of a type *a* we do not yet know. If we use the operators to concatenate lists of numbers, then the *a* in `[a]` will be some type of number (for example, integers). If we are concatenating lists of characters, then *a* represents a Char because String is `[Char]`. The type variable *a* in `[a]` is polymorphic. Polymorphism is an important feature of Haskell. For concatenation, every list must be the same type of list; we cannot concatenate a list of numbers with a list of characters, for example. However, since the *a* is a variable at the type level, the literal values in each list we concatenate need not be the same, only the same type. In other words, *a* must equal *a* (a == a).

```
Prelude> "hello" ++ " Chris"
"hello Chris"

-- but

Prelude> "hello" ++ [1, 2, 3]

<interactive>:14:13:
    No instance for (Num Char) arising from the literal ‘1’
    In the expression: 1
    In the second argument of ‘(++)’, namely ‘[1, 2, 3]’
    In the expression: "hello" ++ [1, 2, 3]
```

In the first example, we have two strings, so the type of *a* matches — they're both Char in `[Char]`, even though the literal values are different. Since the

type matches, no type error occurs and we see the concatenated result. In the second example, we have two lists (a String and a list of numbers) whose types do not match, so we get the error message. GHCi asks for an instance of the numeric typeclass `Num` for the type Char. Unsurprisingly, Char isn't an instance of `Num`.

Intermission: Exercises

Read the syntax of the following functions and decide whether it will compile. Test them in your REPL and try to fix the syntax errors where they occur.

1. `++ [1, 2, 3] [4, 5, 6]`
2. `'<3' ++ ' Haskell'`
3. `concat ["<3", " Haskell"]`

3.5 An example of concatenation and scoping

We will use parentheses to call `++` as a typical prefix (not infix) function:

```
-- print3flipped.hs
module Print3Flipped where

myGreeting :: String
myGreeting = (++) "hello" " world!"

hello :: String
hello = "hello"

world :: String
world = "world!"

main :: IO ()
main = do
  putStrLn myGreeting
  putStrLn secondGreeting
  where secondGreeting = (++) hello ((++) " " world)
  -- could've been: secondGreeting = hello ++ " " ++ world
```

In `myGreeting`, we moved `++` to the front like a typical prefix function invocation. Using it as a prefix function in the `secondGreeting`, though, forces us to shift some things around. Parenthesizing it that way emphasizes the right associativity of the `++` function.

The `where` clause creates local bindings for expressions that are not visible at the top level. In other words, the `where` clause in the main function introduces a definition visible only within the expression or function it's attached to, rather than making it globally visible to the entire module. Something visible at the top level is in scope for all parts of the module and may be exported by the module or imported by a different module. Local definitions, on the other hand, are only visible to that one function. You cannot import into a different module and reuse `secondGreeting`.

To illustrate:

```
-- print3broken.hs
module Print3Broken where

printSecond :: IO ()
printSecond = do
  putStrLn greeting

main :: IO ()
main = do
  putStrLn greeting
  printSecond
  where greeting = "Yarrrrr"
```

You should get an error like this:

```
Prelude> :l print3broken.hs
[1 of 1] Compiling Print3Broken     ( print3broken.hs, interpreted )

print3broken.hs:6:12: Not in scope: ‘greeting’
Failed, modules loaded: none.
```

Let's take a closer look at this error:

```
print3broken.hs:6:12: Not in scope: ‘greeting’
#              [1][2]     [3]          [4]
```

1. The line the error occurred on: in this case, line 6.
2. The column the error occurred on: column 12. Text on computers is often described in terms of lines and columns. These line and column numbers are about lines and columns in your text file containing the source code.
3. The actual problem. We refer to something not in scope, that is, not *visible* to the `printSecond` function.
4. The thing we referred to that isn't visible or in *scope*.

Now make the `Print3Broken` code compile. It should print "Yarrrrr" twice on two different lines and then exit.

3.6 More list functions

You can use most functions for manipulating lists on strings as well in Haskell, because a string is just a list of characters, `[Char]`, under the hood.

Here are some examples:

```
Prelude> :t 'c'
'c' :: Char
Prelude> :t "c"
"c" :: [Char] -- List of Char is String, same thing.

-- the : operator, called "cons," builds a list
Prelude> 'c' : "hris"
"chris"
Prelude> 'P' : ""
"P"

-- head returns the head or first element of a list
Prelude> head "Papuchon"
'P'

-- tail returns the list with the head chopped off
Prelude> tail "Papuchon"
"apuchon"

-- take returns the specified number of elements
-- from the list, starting from the left:
Prelude> take 1 "Papuchon"
"P"
Prelude> take 0 "Papuchon"
""
```

```
Prelude> take 6 "Papuchon"
"Papuch"

-- drop returns the remainder of the list after the
-- specified number of elements has been dropped:
Prelude> drop 4 "Papuchon"
"chon"
Prelude> drop 9001 "Papuchon"
""
Prelude> drop 1 "Papuchon"
"apuchon"

-- we've already seen the ++ operator
Prelude> "Papu" ++ "chon"
"Papuchon"
Prelude> "Papu" ++ ""
"Papu"

-- !! returns the element that is in the specified
-- position in the list. Note that this is an
-- indexing function, and indices in Haskell start
-- from 0. That means the first element of your
-- list is 0, not 1, when using this function.
Prelude> "Papuchon" !! 0
'P'
Prelude> "Papuchon" !! 4
'c'
```

3.7 Chapter Exercises

Reading syntax

1. For the following lines of code, read the syntax carefully and decide if they are written correctly. Test them in your REPL after you've decided to check your work. Correct as many as you can.

a) `concat [[1, 2, 3], [4, 5, 6]]`

b) `++ [1, 2, 3] [4, 5, 6]`

c) `(++) "hello" " world"`

d) `["hello" ++ " world]`

e) `4 !! "hello"`

f) `(!!) "hello" 4`

g) `take "4 lovely"`

h) `take 3 "awesome"`

2. Next we have two sets: the first set is lines of code and the other is a set of results. Read the code and figure out which results came from which lines of code. Be sure to test them in the REPL.

a) `concat [[1 * 6], [2 * 6], [3 * 6]]`

b) `"rain" ++ drop 2 "elbow"`

c) `10 * head [1, 2, 3]`

d) `(take 3 "Julie") ++ (tail "yes")`

e)
```
concat [tail [1, 2, 3],
        tail [4, 5, 6],
        tail [7, 8, 9]]
```

Can you match each of the previous expressions to one of these results presented in a scrambled order?

a) `"Jules"`

b) `[2,3,5,6,8,9]`

c) `"rainbow"`

d) `[6,12,18]`

e) `10`

Building functions

1. Given the list-manipulation functions mentioned in this chapter, write functions that take the following inputs and return the expected outputs. Do them directly in your REPL and use the `take` and `drop` functions you've already seen.

   Example

   ```
   -- If you apply your function to this value:
   "Hello World"
   -- Your function should return:
   "ello World"
   ```

   The following would be a fine solution:

   ```
   Prelude> drop 1 "Hello World"
   "ello World"
   ```

   Now write expressions to perform the following transformations, just with the functions you've seen in this chapter. You do not need to do anything clever here.

   a)
   ```
   -- Given
   "Curry is awesome"
   -- Return
   "Curry is awesome!"
   ```

   b)
   ```
   -- Given:
   "Curry is awesome!"
   -- Return:
   "y"
   ```

   c)
   ```
   -- Given:
   "Curry is awesome!"
   -- Return:
   "awesome!"
   ```

2. Now take each of the above and rewrite it in a source file as a general function that could take different string inputs as arguments but retain the same behavior. Use a variable as the argument to your (named) functions. If you're unsure how to do this, refresh your memory by looking at the `waxOff` exercise from the previous chapter and the `GlobalLocal` module from this chapter.

3. Write a function of type `String -> Char` which returns the third character in a String. Remember to give the function a name and apply it to a variable, not a specific String, so that it could be reused for different String inputs, as demonstrated (feel free to name the function something else. Be sure to fill in the type signature and fill in the function definition after the equals sign):

```
thirdLetter ::
thirdLetter x =

-- If you apply your function to this value:
"Curry is awesome"
-- Your function should return
`r'
```

   Note that programming languages conventionally start indexing things by zero, so getting the zeroth index of a string will get you the first letter. Accordingly, indexing with 3 will actually get you the fourth. Keep this in mind as you write this function.

4. Now change that function so the string input is always the same and the variable represents the number of the letter you want to return (you can use "Curry is awesome!" as your string input or a different string if you prefer).

```
letterIndex  :: Int -> Char
letterIndex x =
```

5. Using the `take` and `drop` functions we looked at above, see if you can write a function called `rvrs` (an abbreviation of 'reverse' used because there is a function called 'reverse' already in Prelude, so if you call your function the same name, you'll get an error message).

`rvrs` should take the string "Curry is awesome" and return the result "awesome is Curry." This may not be the most lovely Haskell code you will ever write, but it is quite possible using only what we've learned so far. First write it as a single function in a source file. This doesn't need, and shouldn't, work for reversing the words of *any* sentence. You're expected only to slice and dice this particular string with `take` and `drop`.

6. Let's see if we can expand that function into a module. Why would we want to? By expanding it into a module, we can add more functions later that can interact with each other. We can also then export it to other modules if we want to and use this code in those other modules. There are different ways you could lay it out, but for the sake of convenience, we'll show you a sample layout so that you can fill in the blanks:

```
module Reverse where

rvrs :: String -> String
rvrs x =

main :: IO ()
main = print ()
```

Into the parentheses after `print` you'll need to fill in your function name `rvrs` plus the argument you're applying `rvrs` to, in this case "Curry is awesome." That `rvrs` function plus its argument are now the argument to `print`. It's important to put them inside the parentheses so that that function gets applied and evaluted first, and then that result is printed.

Of course, we have also mentioned that you can use the $ symbol to avoid using parentheses, too. Try modifying your main function to use that instead of the parentheses.

3.8 Definitions

1. A *String* is a sequence of characters. In Haskell, `String` is represented by a linked-list of `Char` values, aka `[Char]`.

2. A *type* or datatype is a classification of values or data. Types in Haskell determine what values are members of it or *inhabit* it. Unlike in other languages, datatypes in Haskell by default do not delimit the operations that can be performed on that data.

3. *Concatenation* is the joining together of sequences of values. Often in Haskell this is meant with respect to the `[]` or "List" datatype, which also applies to `String` which is `[Char]`. The *concatenation* function in Haskell is `(++)` which has type `[a] -> [a] -> [a]`. For example:

```
Prelude> "tacos" ++ " " ++ "rock"
"tacos rock"
```

4. *Scope* is where a variable referred to by name is valid. Another word used with the same meaning are *visibility*, because if a variable isn't *visible* it's not in *scope*.

5. *Local bindings* are bindings local to particular expression. The primary delineation here from *global* bindings is that *local* bindings cannot be imported by other programs or modules.

6. *Global* or top level bindings in Haskell mean bindings visible to all code within a module and, if made available, can be imported by other modules or programs. Global bindings in the sense that a variable is unconditionally visible throughout an entire program do not exist in Haskell.

7. *Data structures* are a way of organizing data so that the data can be accessed conveniently or efficiently.