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.

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, after all. 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 techniques. 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. Part of human frailty in software is 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 do 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.

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 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.

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 built-in 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'll want to move to text editors that can grow with you over time such as Emacs or Vim.

Chapter 1

Strings

Simple operations with text

Like punning, programming is a play on words

Alan Perlis

1.1 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`s.

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 `String`s
- print strings in the REPL environment
- work with some simple functions that operate on this datatype

A first look at types

First, since we will be working with `String`s, 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 it in the last chapter. Types are quite important in Haskell, and the next chapter is 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 will 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 is telling that "Hello!" has the type `list of Char`.

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`:

```haskell
-- 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           ( print1.hs, interpreted )
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.

```haskell
-- 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           ( print2.hs, interpreted )
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 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
-- print4.hs
module Print4 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 print4.hs
[1 of 1] Compiling Print4           ( print4.hs, interpreted )
Ok, modules loaded: Print4.
*Print4> main
hello world!
hello world!
*Print4>
```

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:

```haskell
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 are how one introduces 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      ( Global.hs, interpreted )
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).

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. Are `r` and `d` in scope for function `area`?

   ```
   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
   ```

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]
```

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 concantenate lists of numbers, then the `a` in `[a]` will be some type of numbers (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`.

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"]`

An example of concatenation and scoping

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

```
-- print4flipped.hs
module Print4Flipped 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:

```
-- print5broken.hs
module Print5Broken 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  print5broken.hs
[1 of 1] Compiling Print5Broken     ( print5broken.hs, interpreted )

print5broken.hs:6:12: Not in scope: ‘greeting’
Failed, modules loaded: none.
Prelude>
```

Let's take a closer look at this error:

```
print5broken.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 `Print5Broken` code compile. It should print "Yarrrrr" twice on two different lines and then exit.

More list functions

You can use most functions for manipulating lists on `String`s 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 just 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'
```

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 previous the 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 before, write a function that takes the following inputs and returns the expected outputs. Do them directly in your REPL:

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

   -- Given:
   "Curry is awesome!"
   -- Return:
   "y"

   --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"
   ```

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.

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.

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 doesn't exist in Haskell.

7. *Data structures* are a way of organizing data so that the data can be accessed conveniently or efficiently. The structure of most data structures in programming are not "self-documenting" and are thus implicit to how the data is accessed.