# Haskell Programming
from first principles

Christopher Allen
Julie Moronuki

Pure functional programming

without fear or frustration

Contents

Chapter 2

Hello, Haskell!

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 necessary tools in order to complete the exercises as you work through the book. At this time, we recommend installing Stack, which will install GHC Haskell, the interactive environment called GHCi, and a project build tool and dependency manager all at once.

You can find the installation instructions online at `http://docs.haskellstack.org/en/stable/README/`, and there is also great documentation that can help you get started using Stack. You can also find installation instructions at `https://github.com/bitemyapp/learnhaskell`, and there you will also find advice on learning Haskell and links to more exercises that may supplement what you're doing with this book.

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 working with 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, 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, type `ghci` or `stack ghci`[1], hit enter, and see something like the following:

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

If you used `stack ghci`[2] there was probably a lot more startup text, and the prompt might be something other than `Prelude`. That's all fine. You may also have a different version of GHC. As long as your GHC version is between 7.8 and 8.0, it should be compatible with everything in this book.

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`.

What is Prelude? Prelude is a library of standard functions. Opening GHCi or Stack GHCi automatically loads those functions so they can be used without needing to do anything special. You can turn Prelude off, as we will show you much later, and there are alternative Preludes, though we won't use them in the book. Prelude is contained in Haskell's `base` package, which can be found at

[1] If you have installed GHC outside of Stack, then you should be able to open it with just the `ghci` command, but if your only GHC installation is what Stack installed, then you will need `stack ghci`.

[2] At this point in the book, you don't need to use `stack ghci`, but in later chapters when we're importing a lot of modules and building projects, it will be much more convenient to use it.

`https://www.stackage.org/package/base`. You'll see us mention sometimes that something or other is "in `base`" which means it's contained in that vast foundational package.

GHCi commands

Throughout the book, we'll be using GHCi commands, such as `:quit` and `:info` in the REPL. Special commands that only GHCi understands begin with the `:` character. `:quit` is *not* Haskell code; it's just a GHCi feature. We will see more of these commands throughout the book.

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 after a few mentions, we will start abbreviating them.

Working from source files

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:

```haskell
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 OK — 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 "Haskell"
Hello, Haskell!
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 "Haskell" (note the quotation marks), and see the output.

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.

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.

We'll cover declarations more later, but it suffices to say for now that they are top-level bindings which allows us to name expressions. We can then use those names 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 by entering the code at the prompt, then hitting '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 get your REPL going to 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, though these are somewhat imprecise.

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 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 *parameters*[3] of the function, separated only by white space. Next there is an equal sign, which expresses equality of the terms. 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:

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

[3] In practice, the terms 'argument' and 'parameter' are often used interchangeably, but there is a difference. 'Argument' properly refers to the value(s) that are passed to the function's parameters when the function is applied, not to the variables that represent them in the function definition (or those in the type signature). See the definitions at the end of the chapter for more information.

Let's examine each part of that:

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

1. This is the name of the function we are defining; it is a function *declaration*. Note that it begins with a lowercase letter.
2. This is the parameter of the function. The parameters of our function correspond to the "head" of a lambda and bind variables that appear in the body expression.
3. The $=$ is used to define (or *declare*) values and functions. Reminder: this is *not* how we test for equality between two values in Haskell.
4. This is the body of the function, an expression that could be evaluated if the function is applied to a value. If `triple` is applied, the argument it's applied to will be the value to which the x is bound. 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! Function names start with lowercase letters. Sometimes for clarity in function names, you may want camelCase style, and that is good style provided the first letter remains lowercase.

Variables must also begin with lowercase letters.

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.

2.5 Evaluation

When we talk about evaluating an expression, we're talking about reducing the terms until the expression 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 nonstrict 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 a value. As in the lambda calculus, application is evaluation: applying a function to an argument allows evaluation or reduction.

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.

Let's get back to our `triple` function. Calling the function by name and applying it to an argument makes it 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; x:= 2]
2 * 3
6
```

We've applied `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.

Haskell doesn't evaluate everything to canonical or normal form by default. Instead, it only evaluates to weak head normal form (WHNF) by default. We'll get into the details of what that means somewhat later in the book. For now, we want to emphasize that Haskell's nonstrict evaluation means not everything will get reduced to its irreducible form immediately, so this:

```
(\f -> (1, 2 + f)) 2
```

reduces to the following in WHNF:

```
(1, 2 + 2)
```

This representation is an approximation, but the key point here is that `2 + 2` is not evaluated to `4` until the last possible moment.

Exercises: Comprehension Check

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)
```

3. There is a value in Prelude called `pi`. Rewrite your function to use `pi` instead of 3.14.

2.6 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 infix style, with a small change in syntax:

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

And you can use infix operators in prefix fashion by wrapping them in parentheses:

```
Prelude> (+) 100 100
200
Prelude> (*) 768395 21356345
16410108716275
Prelude> (/) 123123 123
1001.0
```

If the function name is alphanumeric, it is a prefix function by default, and not all prefix functions can be made infix. If the name is a symbol, it is infix by default but can be made prefix by wrapping it in parentheses.[4]

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 It also tells you whether it's an infix operator,

[4]For people who like nitpicky details: you cannot make a prefix function into an infix function using backticks, then wrap that in parentheses and make it into a prefix function. We're not clear why you'd want to do that anyway. Cut it out.

and, if it is, its associativity and precedence. Let's talk about that associativity and precedence briefly. We will elide the type information and so forth for now.

Here's what the code in Prelude says for `(*)`, `(+)`, and `(-)` at time of writing:

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

:info (+) (-)
infixl 6 +

infixl 6 -
```

1. `infixl` means it's an 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.

The information about addition and subtraction tell us they are both left-associative, infix operators with the same precedence (6).

Let's play with parentheses and see what it means that these associate to the left. 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 with multiplication why associativity mattered, 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?

Exercises: Parentheses and Association

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)`

2.7 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 defined. 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 saw above with the `triple` function, we have to use `let` to declare something in the REPL.

We can now 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:

```haskell
-- 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 variable names. Also, in this variable name, we've used camelCase: 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 whitespace at the end of lines of code, is considered bad style.

In source code files, indentation 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:

```haskell
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:

```haskell
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:

```haskell
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. Indentation can easily go wrong in a copy-and-paste job as well.

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. Most text editors can be configured to use only spaces for indentation, so you may want to investigate how to do that for yours.

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:

```haskell
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 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. 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:

```haskell
-- 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:

```
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
```

Exercises: Heal the Sick

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
   ```

2.8 Arithmetic functions in Haskell

This section will explore some basic arithmetic using some common operators and functions for arithmetic. We'll focus on the following subset of them:

Operator	Name	Purpose/application
+	plus	addition
-	minus	subtraction
*	asterisk	multiplication
/	slash	fractional division
div	divide	integral division, round down
mod	modulo	like 'rem', but after modular 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
```

You will usually want `div` for integral division unless you know what you're doing, due to the way `div` and `quot` round:

```
-- rounds down
Prelude> div 20 (-6)
-4

-- rounds toward zero
Prelude> quot 20 (-6)
-3
```

Also, `rem` and `mod` have slightly different use cases; we'll look at `mod` in a little more detail down below. We will cover `(/)` in more detail in a later chapter, as that will require some explanation of types and typeclasses.

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.[5]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))

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

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

10 == x == yeppers.
```

It's not worth getting overly invested in the meaning of "yeppers" there; it just means we got to the result we wanted.

Now for `div` and `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.
```

Our result indicates all is well in the world of integral division.

Using 'mod'

This section is not a full discussion of modular arithmetic, but we want to give more direction in how to use `mod` in general, for those who may be unfamiliar with it, and how it works in Haskell specifically.

[5] From Lennart Augustsson's blog `http://augustss.blogspot.com/` or Stack Overflow answer at `http://stackoverflow.com/a/8111203`

We've already mentioned that `mod` gives the remainder of a modular division. If you're not already familiar with modular division, you may not understand the useful difference between `mod` and `rem`.

Modular arithmetic is a system of arithmetic for integers where numbers "wrap around" upon reaching a certain value, called the *modulus*. It is often explained in terms of a clock.

When we count time by a 12-hour clock, we have to wrap the counting around the 12. For example, if the time is now 8:00 and you want to know what time it will be 8 hours from now, you don't simply add 8 + 8 and get a result of 16 o'clock.[6]

Instead, you wrap the count around every 12 hours. So, adding 8 hours to 8:00 means that we add 4 hours to get to the 12, and at the 12 we start over again as if it's 0 and add the remaining 4 hours of our 8, for an answer of 4:00. That is, 8 hours after 8:00 is 4:00.

This is arithmetic *modulo* 12. In our 12-hour clock, 12 is equivalent to both itself and to 0, so the time at 12:00 is also, in some sense 0:00. Arithmetic modulo 12 means that 12 is both 12 and 0.

Often, this will give you the same answer that `rem` does:

```
Prelude> mod 15 12
3
Prelude> rem 15 12
3

Prelude> mod 21 12
9
Prelude> rem 21 12
9

Prelude> mod 3 12
3
Prelude> rem 3 12
3
```

If you're wondering what the deal is with the last two examples, it's because `mod` and `rem` can only represent integral division. If all

[6]Obviously, with a 24-hour clock, such a time is possible; however, if we were starting from 8:00 p.m. and trying to find the time 8 hours later, the answer would not be 16:00 a.m. A 24-hour clock has a different modulus than a 12-hour clock.

you have to work with is integers, then dividing a smaller number by a larger number results in an answer of 0 with a remainder of whatever the smaller number (the dividend) is. If you want to divide a smaller number by a larger number and return a fractional answer, then you need to use `(/)`, and you won't have a remainder.

Let's say we need to write a function that will determine what day of the week it was or will be a certain number of days before or after this one. For our purposes here, we will assign a number to each day of the week, using 0 to represent Sunday.[7] Then if today is Monday, and we want to know what day of the week it will be 23 days from now, we could do this:

```
Prelude> mod (1 + 23) 7
3
```

The 1 represents Monday, the current day, while 23 is the number of days we're trying to add. Using `mod` to wrap it around the 7 means it will return a number that corresponds to a day of the week in our numbering.

And 5 days from Saturday will be Thursday:

```
Prelude> mod (6 + 5) 7
4
```

We can use `rem` to do the same thing with apparently equivalent accuracy:

```
Prelude> rem (1 + 23) 7
3
```

However, if we want to *subtract* and find out what day of the week it was some number of days ago, then we'll see a difference. Let's try asking, if today is Wednesday (3), what day it was 12 days ago:

```
Prelude> mod (3 - 12) 7
5
Prelude> rem (3 - 12) 7
-2
```

[7] Sure, you may naturally think of the days of week as being numbered 1-7. But programmers like to index things from zero.

The version with `mod` gives us a correct answer, while the `rem` version does not.

One key difference here is that, in Haskell (not in all languages), if one or both arguments are negative, the results of `mod` will have the same sign as the divisor, while the result of `rem` will have the same sign as the dividend:

```
Prelude> (-5) `mod` 2
1
Prelude> 5 `mod` (-2)
-1
Prelude> (-5) `mod` (-2)
-1

-- but

Prelude> (-5) `rem` 2
-1
Prelude> 5 `rem` (-2)
1
Prelude> (-5) `rem` (-2)
-1
```

Figuring out when you need `mod` takes some experience, and it may not be obvious right now. But you will need it later in the book.

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 thinks we are trying to add and subtract, not add a negative number, so it 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 can make the typing or reading of code nicer but changes nothing about the semantics, or meaning, of programs and doesn't change how we solve 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 syntactic 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 `-` being used for subtraction:

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

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

2.9 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 at this time. 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:

```
Prelude> (2^) (2 + 2)
16
-- can replace those parentheses
Prelude> (2^) $ 2 + 2
16
-- without either parentheses or $
Prelude> (2^) 2 + 2
6
```

The `($)` will allow everything to the right of it to be evaluated first and can be used to delay function application. You'll see what we mean about delaying function application in particular when we get to Chapter 7 and use it with function composition.

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
```

Some Haskellers find parentheses more readable than the dollar sign, but it's too common in idiomatic Haskell code for you to not at least be familiar with it.

Parenthesizing infix operators

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
```

Or you instead of (- x), you can write (subtract x):

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

It may not be immediately obvious why you would ever want to do this, but you will see this syntax used throughout the book, for example, once we start wanting to apply functions to each value inside a list or other data structure. We will discuss partial application of functions in more detail in a later chapter as well.

2.10 Let and where

You will often see `let` and `where` used to introduce components of expressions, and they seem similar. It takes some practice to get used to the appropriate times to use each, but they are fundamentally different.

The contrast here is that `let` introduces an *expression*, so it can be used wherever you can have an expression, but `where` is a *declaration* and is bound to a surrounding syntactic construct.

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

```haskell
-- 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`:

```haskell
-- 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 `FunctionWithLet` in the same REPL session as `FunctionWithWhere`, then it will have unloaded the first one before loading the new one:

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

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 Stack rather than GHCi itself.

Exercises: A Head Code

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 using `where` declarations. 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 things 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:

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

The prompt changes to `*Main` instead of `Prelude` to indicate that you have a module called `Main` loaded.

Rewrite 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`

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

2.11 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

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 an expression 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.

2.12 Definitions

1. The terms *argument* and *parameter* are often used interchangeably. However, it is worthwhile to understand the distinction. A *parameter*, or formal parameter, *represents* a value that will be passed to the function when the function is called. Thus, parameters are usually variables. An *argument* is an input value the function is applied to. A function's parameter is bound to the value of an argument when the function is applied to that argument. For example, in `f x = x + 2` which takes an argument and returns that value added to 2, x is the one parameter of our function. We run the code by applying f to some argument. If the argument we passed to the parameter x were 2, our result

would be 4. However, arguments can themselves be variables or be expressions that include variables, thus the distinction is not always clear. When we use “parameter” in this book, it will always be referring to formal parameters, usually in a type signature, but we’ve taken the liberty of using “argument” somewhat more loosely.

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 the term *expression*.

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.

2.13 Follow-up resources

1. Haskell wiki article on Let vs. Where
   `https://wiki.haskell.org/Let_vs._Where`

2. How to desugar Haskell code; Gabriel Gonzalez

Chapter 3

Strings

Like punning, programming
is a play on words

Alan Perlis

3.1 Printing strings

So far we've been doing arithmetic using simple expressions. In this chapter, we will turn our attention to 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 standard 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 as well as 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. `String` is a *type alias*, or type synonym, for a list of `Char`. A type alias is what it sounds like: we use one name for a type, usually for convenience, that has a different type name underneath. Here `String` is another name for a list of characters. By using the name `String` we are able to visually differentiate it from other types of lists, and names themselves don't mean much to the computer. When we talk about lists in more detail later, we'll see why the square brackets are 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 see how to print 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 load it in GHCi and run `main`:

```
Prelude> :l print1.hs
[1 of 1] Compiling Print1
Ok, modules loaded: Print1.
*Print1> main
hello world!
*Print1>
```

First, note that your `Prelude>` prompt may have changed to reflect the name of the module you loaded. You can use `:module` or `:m` to unload the module and return to Prelude if you wish. You can also set your prompt to something specific, which means it won't change every time you load or unload a module[1]:

```
Prelude> :set prompt "λ> "
λ> :r
Ok, modules loaded: Print1.
λ> main
hello world!
λ>
```

Looking at the code again, `main` is the default action when you build an executable or run it in a REPL. It is not a function but is often a series of instructions to execute, which can include applying functions and producing side-effects. When building a project with Stack, having a `main` executable in a `Main.hs` file is obligatory, but you can have source files and load them in GHCi without necessarily having a `main` block.

As you can see, `main` has the type `IO ()`. `IO` stands for input/output but has a specialized meaning in Haskell. It is a special type used when the result of running the program involves effects in addition to being a function or expression. Printing to the screen is an 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` action is the default executable, you will see it 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:

[1]You can set it permanently if you prefer by setting the configuration in your /.ghci 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!"
```

This `do` syntax is a special syntax that allows for sequencing actions. It is most commonly used to sequence the actions that constitute your program, some of which will necessarily perform effects such as printing to the screen (that's why the obligatory type of `main` is `IO ()`). The `do` isn't strictly necessary, but since it often makes for more readable code than the alternatives, you'll see it a lot. We will explain it a bit more in Chapter 13, and there will be a full explanation in the chapter on Monad.

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

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]
```

Remember, `String` is a type synonym for `[Char]`. You can try changing the type signatures to reflect that and see if it changes anything in the program execution.

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>
```

This little exercise demonstrates a few things:

1. We defined values at the top level of a module: (`myGreeting`, `hello`, `world`, and `main`). That is, they were declared at the top level so that they are available throughout the module.
2. We specify explicit types for top-level definitions.
3. We concatenate strings with `(++)` and `concat`.

3.4 Top-level versus local definitions

What does it mean for something to be at the top level of a module? It doesn't necessarily mean it's defined at the top of the file. When the compiler reads the file, it will see all the top-level declarations, no matter what order they come in the file (with some limitations which we'll see later). Top-level declarations are not nested within anything else and they are in scope throughout the whole module.

We can contrast a top-level definition with a local definition. To be locally defined would mean the declaration is nested within some other expression and is not visible outside that expression. We practiced this in the previous chapter with `let` and `where`. Here's an example for review:

```
module TopOrLocal 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, and they are accessible to everything else in the 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 less repetitious.

Also note we explicitly declared the type of `woot` in the `where` clause, using the :: syntax. This wasn't necessary (Haskell's type inference would've figured it out), but it was done here to show you how. Be sure to load and run this code in your REPL:

```
Prelude> :l TopOrLocal.hs
[1 of 1] Compiling TopOrLocal
Ok, modules loaded: TopOrLocal.
*TopOrLocal> 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: Scope

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? Go with your gut here.

   ```
   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
   ```

3.5 Types 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]
```

The type of `concat` says that we have a list of lists as input and we will return a list. It will have the same values inside it as the list of lists did; it just flattens it into one list structure, in a manner of speaking. A `String` is a list, a list of `Char` specifically, and `concat` can work on lists of strings or lists of lists of other things:

```
Prelude> concat [[1, 2], [3, 4, 5], [6, 7]]
[1,2,3,4,5,6,7]
Prelude> concat ["Iowa", "Melman", "Django"]
"IowaMelmanDjango"
```

(n.b., Assuming you are using GHC 7.10 or higher, if you check this type signature in your REPL, you will find different 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]`. This type is a list of elements of some type a. This function does not know what type a is. It doesn't need to know. In the context of the program, the type of a will be known and made concrete at some point.
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`. We will discuss typeclasses later. Typeclasses provide definitions of operations, or functions, that can be shared across sets of types. For now, you can understand this message as telling you the types don't match so it can't concatenate the two lists.

Exercises: Syntax Errors

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"]`

3.6 Concatenation and scoping

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

```haskell
-- 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 `secondGreeting`, using `++` as a prefix function forces us to shift some things around. Parenthesizing it that way emphasizes the right associativity of the `++` function. Since it's an infix operator, you can check for yourself that it's right associative:

```
Prelude> :i (++)
(++) :: [a] -> [a] -> [a]   -- Defined in ‘GHC.Base’
infixr 5 ++
```

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 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.

3.7 More list functions

Since a `String` is a specialized type of list, you can use standard list operations on strings as well.

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'
```

Note that while all these functions are standard Prelude functions, many of them are considered *unsafe* They are unsafe because they do not cover the case where they are given an empty list as input. Instead they just throw out an error message, or *exception*:

```
Prelude> head ""
*** Exception: Prelude.head: empty list
Prelude> "" !! 4
*** Exception: Prelude.!!: index too large
```

This isn't ideal behavior, so the use of these functions is considered unwise for programs of any real size or complexity, although we will use them in these first few chapters. We will address how to cover all cases and make safer versions of such functions in a later chapter.

3.8 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 `TopOrLocal` 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 operated on 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 to, 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.

3.9 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 the type or that

inhabit the type. 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 is *visibility*, because if a variable isn't *visible* it's not in *scope*.

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

6. *Top level bindings* in Haskell are bindings that stand outside of any other declaration. The main feature of top-level bindings is that they can be made available to other modules within your programs or to other people's programs.

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

Chapter 4

Basic datatypes

There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.

Robin Milner

4.1 Basic Datatypes

Haskell has a robust and expressive type system. Types play an important role in the readability, safety, and maintainability of Haskell code as they allow us to classify and delimit data, thus restricting the forms of data our programs can process. Types, also calld *datatypes*, provide the means to build programs more quickly and also allow for greater ease of maintenance. As we learn more Haskell, we'll learn how to leverage types in a way that lets us accomplish the same things but with *less* code.

So far, we've looked at expressions involving numbers, characters, and lists of characters, also called strings. These are some of the standard built-in datatypes that categorize and delimit values. While those are useful datatypes and cover a lot of types of values, they certainly don't cover every type of data. In this chapter, we will

- review types we have seen in previous chapters;
- learn about datatypes, type constructors, and data constructors;
- work with predefined datatypes;
- learn more about type signatures and a bit about typeclasses.

4.2 What are types?

Expressions, when evaluated, reduce to values. Every value has a type. Types are how we group a set of values together that share something in common. Sometimes that "something in common" is abstract, sometimes it's a specific model of a particular concept or domain. If you've taken a mathematics course that covered sets, thinking about types as being like sets will help guide your intuition on what types are and how they work in a mathematical[1] sense.

[1] Set theory is the study of mathematical collections of objects. Set theory was a precursor to type theory, the latter being used prolifically in the design of programming languages like Haskell. Logical operations like disjunction (or) and conjunction (and) used in the manipulation of sets have equivalents in Haskell's type system.

4.3 Anatomy of a data declaration

Data declarations are how datatypes are defined.

The type constructor is the name of the type and is capitalized. When you are reading or writing type signatures (the type level of your code), the type names or type constructors are what you use.

Data constructors are the values that inhabit the type they are defined in. They are the values that show up in your code, at the term level instead of the type level. By "term level", we mean they are the values as they appear in your code or the values that your code evaluates to.

We will start with a basic datatype to see how datatypes are structured and get acquainted with the vocabulary. `Bool` isn't a datatype we've seen yet in the book, but it provides for truth values. It is named after the great logician George Boole and the system of logic named for him. Because there are only two truth values, it's easy to list all the data constructors:

```
-- the definition of Bool
data Bool = False | True
--    [1]    [2] [3] [4]
```

1. Type constructor for datatype `Bool`. This is the name of the type and shows up in type signatures.

2. Data constructor for the value `False`.

3. Pipe `|` indicates a *sum type* or logical disjunction: "or." So, a `Bool` value is `True` *or* `False`.

4. Data constructor for the value `True`.

The whole thing is called a data declaration. Data declarations do not always follow precisely the same pattern — there are datatypes that use logical conjunction ("and") instead of disjunction, and some type constructors and data constructors may have arguments. The thing they have in common is the keyword `data` followed by the type constructor (or name of the type that will appear in type signatures), the equals sign to denote a definition, and then data constructors (or names of values that inhabit your term-level code).

You can find the datatype definition for built-in datatypes by using `:info` in GHCi:

```
Prelude> :info Bool
data Bool = False | True
```

Let's look at where different parts of datatypes show up in our code. If we query the type information for a function called `not` we see that it takes one `Bool` value and returns another `Bool` value, so the type signature makes reference to the type constructor, or datatype name:

```
Prelude> :t not
not :: Bool -> Bool
```

But when we use the `not` function, we use the data constructors, or values:

```
Prelude> not True
False
```

And our expression evaluates to another data constructor, or value — in this case the other data constructor for the same datatype.

Exercises: Mood Swing

Given the following datatype, answer the following questions:

```
data Mood = Blah | Woot deriving Show
```

The `deriving Show` part is not something we've explained yet. For now, all we'll say is that when you make your own datatypes, deriving `Show` allows the values of that type to be printed to the screen. We'll talk about it more when we talk about typeclasses in detail.

1. What is the type constructor, or name of this type?
2. If the function requires a `Mood` value, what are the values you could possibly use there?

3. We are trying to write a function `changeMood` to change Chris's mood instantaneously. It should act like `not` in that, given one value, it returns the *other* value of the same type. So far, we've written a type signature `changeMood :: Mood -> Woot`. What's wrong with that?

4. Now we want to write the function that changes his mood. Given an input mood, it gives us the other one. Fix any mistakes and complete the function:

```
changeMood Mood = Woot
changeMood    _ = Blah
```

   We're doing something here called *pattern matching*. We can define a function by matching on a data constructor, or value, and descrbing the behavior the function should have based on which value it matches. The underscore in the second line denotes a catch-all, otherwise case. So, in the first line of the function, we're telling it what to do in the case of a specific input. In the second one, we're telling it what to do regardless of *all potential inputs*. It's trivial when there are only two potential values of a given type, but as we deal with more complex cases, it can be convenient. We'll talk about pattern matching in greater detail in a later chapter.

5. Enter all of the above — datatype (including the `deriving Show` bit), your corrected type signature, and the corrected function into a source file. Load it and run it in GHCi to make sure you got it right.

4.4 Numeric types

Let's next look at numeric types, because we've already seen these types in previous chapters, and numbers are familiar territory. It's important to understand that Haskell does not just use one type of number. For most purposes, the types of numbers we need to be concerned with are:

Integral numbers These are whole numbers, positive and negative.

1. `Int`: This type is a fixed-precision integer. By "fixed precision," we mean it has a range, with a maximum and a minimum, and so it cannot be arbitrarily large or small — more about this in a moment.

2. `Integer`: This type is also for integers, but this one supports arbitrarily large (or small) numbers.

Fractional These are not integers. `Fractional` values include the following four types:

1. `Float`: This is the type used for single-precision floating point numbers. Fixed-point number representations have immutable numbers of digits assigned for the parts of the number before and after the decimal point. In contrast, floating point can shift how many bits it uses to represent numbers before or after the decimal point. This flexibility does, however, mean that floating point arithmetic violates some common assumptions and should only be used with great care. Generally, floating point numbers should not be used at all in business applications.

2. `Double`: This is a double-precision floating point number type. It has twice as many bits with which to describe numbers as the `Float` type.

3. `Rational`: This is a fractional number that represents a ratio of two integers. The value `1 / 2 :: Rational` will be a value carrying two `Integer` values, the numerator 1 and the denominator 2, and represents a ratio of 1 to 2. `Rational` is arbitrarily precise but not as efficient as `Scientific`.

4. `Scientific`: This is a space efficient and almost-arbitrary precision scientific number type. `Scientific` numbers are represented using scientific notation. It stores the coefficient as an `Integer` and the exponent as an `Int`. Since `Int` isn't arbitrarily-large there is technically a limit to the size of number you can express with `Scientific`, but hitting that limit is quite unlikely. `Scientific` is available in a library[2]

[2]Hackage page for *scientific* `https://hackage.haskell.org/package/scientific` and can be installed using `cabal install` or `stack install`.

These numeric datatypes all have instances of a typeclass called `Num`. We will talk about typeclasses in the upcoming chapters, but as we look at the types in this section, you will see `Num` listed in some of the type information.

Typeclasses are a way of adding functionality to types that is reusable across all the types that have instances of that typeclass. `Num` is a typeclass for which most numeric types will have an instance because there are standard functions that are convenient to have available for all types of numbers. The `Num` typeclass is what provides your standard `(+)`, `(-)`, and `(*)` operators along with a few others. Any type that has an instance of `Num` can be used with those functions. An instance defines how the functions work for that specific type. We will talk about typeclasses in much more detail soon.

Integral numbers

As we noted above, there are two main types of integral numbers: `Int` and `Integer`.

Integral numbers are whole numbers with no fractional component. The following are integral numbers:

```
1 2 199 32442353464675685678
```

The following are not integral numbers:

```
1.3 1/2
```

Integer

The numbers of type `Integer` are straightforward enough; for the most part, they are the sorts of numbers we're all used to working with in arithmetic equations that involve whole numbers. They can be positive or negative, and `Integer` extends as far in either direction as one needs them to go.

The `Bool` datatype only has two possible values, so we can list them explicitly as data constructors. In the case of `Integer`, and most numeric datatypes, these data constructors are not written out because they include an infinite series of whole numbers. We'd need infinite data constructors stretching up and down from zero. Hypothetically we could represent `Integer` as a sum of three cases, recursive constructors headed towards negative infinity, zero, and recursive constructors headed towards positive infinity. This representation would be terribly inefficient so there's some GHC magic sprinkled on it.

Why do we have Int?

The `Int` numeric type is an artifact of what computer hardware has supported natively over the years. Most programs should use `Integer` and not `Int`, unless the limitations of the type are understood and the additional performance makes a difference.

The danger of `Int` and the related types `Int8`, `Int16`, et al. is that they cannot express arbitrarily large quantities of information. Since they are integral, this means they cannot be arbitrarily large in the positive or negative sense.

Here's what happens if we try to represent a number too large for `Int8`:

```
Prelude> import GHC.Int
Prelude> 127 :: Int8
127
Prelude> 128 :: Int8

<interactive>:11:1: Warning:
    Literal 128 is out of the Int8 range -128..127
    If you are trying to write a large negative literal,
    use NegativeLiterals
-128
Prelude> (127 + 1) :: Int8
-128
```

The syntax you see there, `:: Int8` is us assigning the `Int8` type to these numbers. As we will explain in more detail in the next chapter, numbers are polymorphic underneath and the compiler doesn't assign them a concrete type until it is forced to. It would be weird and unexpected if the compiler defaulted all numbers to `Int8`, so in order to reproduce the situation of having a number too large for an `Int` type, we had to assign that concrete type to it.

The first computation is fine, because it is within the range of valid values of type `Int8`, but the `127 + 1` overflows and resets back to its smallest numeric value. Because the memory the value is allowed to occupy is fixed for `Int8`, it cannot grow to accommodate the value 128 the way `Integer` can. Here the 8 represents how many bits the type uses to represent integral numbers.[3] Being of a fixed size can be useful in some applications, but most of the time `Integer` is preferred.

You can find out the minimum and maximum bounds of numeric types using `maxBound` and `minBound` from the `Bounded` typeclass. Here's an example using our `Int8` and `Int16` example:

```
Prelude> import GHC.Int
Prelude> :t minBound
minBound :: Bounded a => a
Prelude> :t maxBound
maxBound :: Bounded a => a

Prelude> minBound :: Int8
-128
Prelude> minBound :: Int16
-32768
Prelude> minBound :: Int32
-2147483648
Prelude> minBound :: Int64
-9223372036854775808

Prelude> maxBound :: Int8
127
Prelude> maxBound :: Int16
32767
Prelude> maxBound :: Int32
2147483647
Prelude> maxBound :: Int64
9223372036854775807
```

Thus you can find the limitations of possible values for any type that has an instance of that particular typeclass. In this case, we are able to find out the range of values we can represent with an `Int8` is -128 to 127.

[3]The representation used for the fixed-size `Int` types is *two's complement*.

You can find out if a type has an instance of `Bounded`, or any other typeclass, by asking GHCi for the `:info` for that type. Doing this will also give you the datatype representation for the type you queried:

```
Prelude> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
instance Bounded Int -- Defined in 'GHC.Enum'
```

`Int` of course has many more typeclass instances, but `Bounded` is the one we cared about at this time.

Fractional numbers

The four common `Fractional` types in use in Haskell are `Float`, `Double`, `Rational`, and `Scientific`. `Rational`, `Double`, and `Float` come with your install of GHC. `Scientific` comes from a library, as we mentioned previously. `Rational` and `Scientific` are arbitrary precision, with the latter being much more efficient. Arbitrary precision means that these can be used to do calculations requiring a high degree of precision rather than being limited to a specific degree of precision, the way `Float` and `Double` are. You almost never want a `Float` unless you're doing graphics programming such as with OpenGL.

Some computations involving numbers will be fractional rather than integral. A good example of this is the division function `(/)` which has the type:

```
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a
```

The notation `Fractional a =>` denotes a typeclass constraint. You can read it as "the type variable a must implement the `Fractional` typeclass." This tells us that whatever type of number a turns out to be, it must be a type that has an instance of the `Fractional` typeclass; that is, there must be a declaration of how the operations from that typeclass will work for the type. The `/` function will take one number that implements `Fractional`, divide it by another of the same type, and return a value of the same type as the result.

`Fractional` is a typeclass that requires types to already have an instance of the `Num` typeclass. We describe this relationship between typeclasses by saying that `Num` is a superclass of `Fractional`. So `(+)` and other functions from the `Num` typeclass can be used with `Fractional` numbers, but functions from the `Fractional` typeclass cannot be used with all types that have a `Num` instance:

Here's what happens when we use `(/)` in the REPL:

```
Prelude> 1 / 2
0.5
Prelude> 4 / 2
2.0
```

Note that even when we had a whole number as a result, the number was printed as `2.0`, despite having no fractional component. This is because values of `Fractional a => a` default to the floating point type `Double`. In most cases, you won't want to explicitly use `Double`. You're usually better off using the arbitrary precision sibling to `Integer`, `Scientific`. Most people do not find it easy to reason about floating point arithmetic and find it difficult to code around the quirks (those quirks exist by design, but that's another story), so in order to avoid making mistakes, use arbitrary-precision types as a matter of course.

4.5 Comparing values

Up to this point, most of our operations with numbers have involved doing simple arithmetic. We can also compare numbers to determine whether they are equal, greater than, or less than:

```
Prelude> let x = 5
Prelude> x == 5
True
Prelude> x == 6
False
Prelude> x < 7
True
Prelude> x > 3
True
Prelude> x /= 5
False
```

Notice here that we first declared a value for x using the standard equals sign. Now we know that for the remainder of our REPL session, all instances of x will be the value 5. Because the equals sign in Haskell is already used to define variables and functions, we must use a double equals sign, `==`, to have the specific meaning "is equal to." The `/=` symbol means "is not equal to." The other symbols should already be familiar to you.

Having done this, we see that GHCi is returning a result of either `True` or `False`, depending on whether the expression is true or false. `True` and `False` are the data constructors for the `Bool` datatype we saw above. If you look at the type information for any of these infix operators, you'll find the result type listed as `Bool`:

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
```

Notice that we get some typeclass constraints again. `Eq` is a typeclass that includes everything that can be compared and determined to be equal in value; `Ord` is a typeclass that includes all things that can be ordered. Note that neither of these is limited to numbers. Numbers can be compared and ordered, of course, but so can letters, so this typeclass constraint allows for flexibility. These equality and comparison functions can take any values that can be said to have equal value or can be ordered. The rest of the type information tells us that it takes one of these values, compares it to another value of the same type, and returns a `Bool` value. As we've already seen, the `Bool` values are `True` or `False`.

With this information, let's try playing with some other values:

```
Prelude> 'a' == 'a'
True
Prelude> 'a' == 'b'
False
Prelude> 'a' < 'b'
True
Prelude> 'a' > 'b'
False
```

```
Prelude> 'a' == 'A'
False
Prelude> "Julie" == "Chris"
False
```

These examples are easy enough to understand. We know that alphabetical characters can be ordered, although we do not normally think of 'a' as being "less than" 'b.' But we can understand that here it means *'a' comes before 'b'* in alphabetical order. Further, we see this also works with strings such as "Julie" or "Chris." GHCi has faithfully determined that those two strings are not equal in value.

Now use your REPL to determine whether 'a' or 'A' is greater.

Next, take a look at this sample and see if you can figure out why GHCi returns the given results:

```
Prelude> "Julie" > "Chris"
True
Prelude> "Chris" > "Julie"
False
```

Good to see Haskell code that reflects reality. "Julie" is greater than "Chris" because J > C, if the words had been "Back" and "Brack" then it would've skipped the first letter to determine which was greater because B == B, then "Brack" would've been greater because 'r' > 'a' in the lexicographic ordering Haskell uses for characters. Note that this is leaning on the `Ord` typeclass instances for list *and* `Char`. You can only compare lists of items where the items themselves also have an instance of `Ord`. Accordingly, the following will work because `Char` and `Integer` have instances of `Ord`:

```
Prelude> ['a', 'b'] > ['b', 'a']
False
Prelude> 1 > 2
False
Prelude> [1, 2] > [2, 1]
False
```

A datatype that has no instance of `Ord` will not work with these functions:

```
Prelude> data Mood = Blah | Woot deriving Show
Prelude> [Blah, Woot]
[Blah,Woot]
Prelude> [Blah, Woot] > [Woot, Blah]

<interactive>:28:14:
    No instance for (Ord Mood) arising
      from a use of ‘>’
    In the expression: [Blah, Woot] > [Woot, Blah]
    In an equation for ‘it’:
      it = [Blah, Woot] > [Woot, Blah]
```

"No instance for `(Ord Mood)`" means it doesn't have an `Ord` instance to know how to order these values.

Here is another thing that doesn't work with these functions:

```
Prelude> "Julie" == 8

<interactive>:38:12:
    No instance for (Num [Char]) arising from
    the literal ‘8’

    In the second argument of ‘(==)’, namely ‘8’
    In the expression: "Julie" == 8
    In an equation for ‘it’: it = "Julie" == 8
```

We said above that comparison functions are polymorphic and can work with a lot of different types. But we also noted that the type information only admitted values of matching types. Once you've given a term-level value that is a string such as "Julie," the type is determined and the other argument must have the same type. The error message we see above is telling us that the type of the literal value '8' doesn't match the type of the first value, and for this function, it must.

4.6 Go on and Bool me

In Haskell the `Bool` datatype comes standard in the Prelude. As we saw earlier, `Bool` is a sum type with two constructors:

```
data Bool = False | True
```

This declaration creates a datatype with the type constructor `Bool`, and we refer to specific types by their type constructors. We use type constructors in type signatures, not in the expressions that make up our term-level code. The type constructor `Bool` takes no arguments (some type constructors do take arguments). The definition of `Bool` above also creates two data constructors, `True` and `False`. Both of these values are of type `Bool`. Any function that accepts values of type Bool must allow for the possibility of `True` *or* `False`; you cannot specify in the type that it should only accept one specific value. An English language formulation of this datatype would be something like: "The datatype `Bool` is represented by the values True or False."

Remember, you can find the type of any value by asking for it in GHCi, just as you can with functions:

```
Prelude> :t True
True :: Bool
Prelude> :t "Julie"
"Julie" :: [Char]
```

Now let's have some fun with `Bool`. We'll start by reviewing the `not` function:

```
Prelude> :t not
not :: Bool -> Bool
Prelude> not True
False
```

Note that we capitalize `True` and `False` because they are our data constructors. What happens if you try to use `not` without capitalizing them?

Let's try something slightly more complex:

```
Prelude> let x = 5
Prelude> not (x == 5)
False
Prelude> not (x > 7)
True
```

We know that comparison functions evaluate to a `Bool` value, so we can use them with `not`.

Let's play with infix operators that deal directly with boolean logic. How do we use `Bool` and these associated functions?

```
-- && is conjunction, so
-- this means "true and true."
Prelude> True && True
True
-- || is disjunction, so
-- this means "false or true."
Prelude> False || True
True
Prelude> True && False
False
Prelude> not True
False
Prelude> not (True && True)
False
```

We can look up info about datatypes that are in scope (if they're not in scope, we have to import the module they live in to bring them into scope) using the `:info` command GHCi provides. Scope is a way to refer to where a named binding to an expression is valid. When we say something is "in scope", it means you can use that expression by its bound name, either because it was defined in the same function or module, or because you imported it. So, it's visible to the program we're trying to run right now. What is built into Haskell's `Prelude` module is significant because everything in it is automatically imported and in scope. We will demonstrate how to shut this off later, but for now, this is what you want.

Exercises: Find the Mistakes

The following lines of code may have mistakes — some of them won't compile! You know what you need to do.

1. `not True && true`

2. `not (x = 6)`

3. `(1 * 2) > 5`
4. `[Merry] > [Happy]`
5. `[1, 2, 3] ++ "look at me!"`

Conditionals with if-then-else

Haskell doesn't have 'if' statements, but it does have *if expressions*. It's a built-in bit of syntax that works with the `Bool` datatype.

```
Prelude> if True then "Truthin'" else "Falsin'"
"Truthin'"
Prelude> if False then "Truthin'" else "Falsin'"
"Falsin'"
Prelude> :t if True then "Truthin'" else "Falsin'"
if True then "Truthin'" else "Falsin'" :: [Char]
```

The structure here is:

```
if CONDITION
then EXPRESSION\_A
else EXPRESSION\_B
```

If the condition (which must evaluate to `Bool`) reduces to the Bool value `True`, then `EXPRESSION_A` is the result, otherwise `EXPRESSION_B`. Here the type was `String` (aka `[Char]`) because that's the type of the value that is returned as a result.

If-expressions can be thought of as a way to choose between two values. You can embed a variety of expressions within the `if` of an if-then-else, as long as it evaluates to `Bool`. The types of the expressions in the `then` and `else` clauses must be the same, as in the following:

```
Prelude> let x = 0
Prelude> if (x + 1 == 1) then "AWESOME" else "wut"
"AWESOME"
```

Here's how it reduces:

```
-- Given:
x = 0

if (x + 1 == 1) then "AWESOME" else "wut"
-- x is zero

if (0 + 1 == 1) then "AWESOME" else "wut"
-- reduce 0 + 1 so we can see if it's equal to 1

if (1 == 1) then "AWESOME" else "wut"
-- Does 1 equal 1?

if True then "AWESOME" else "wut"
-- pick the branch based on the Bool value

"AWESOME"
-- dunzo
```

But this does not work:

```
Prelude> let x = 0
Prelude> if x * 100 then "adopt a dog" else "or a cat"

<interactive>:15:7:
    No instance for (Num Bool) arising
    from a use of ‘*’

    In the expression: (x * 100)
    In the expression:
       if (x * 100)
         then "adopt a dog"
         else "or a cat"
    In an equation for ‘it’:
        it = if (x * 100)
               then "adopt a dog"
               else "or a cat"
```

We got this type error because the condition passed to the `if` expression is of type `Num a => a`, not `Bool` and `Bool` doesn't implement the `Num` typeclass. To oversimplify, `(x * 100)` evaluates to a numeric result, and numbers aren't truth values. It would have typechecked had the condition been `x * 100 == 0` or `x * 100 == 9001`. In those cases, it would've been an equality check of two numbers which reduces to a Bool value.

Here's an example of a function that uses a `Bool` value in an if expression:

```
-- greetIfCool1.hs
module GreetIfCool1 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool = coolness == "downright frosty yo"
```

When you test this in the REPL, it should play out like this:

```
Prelude> :l greetIfCool1.hs
[1 of 1] Compiling GreetIfCool1
Ok, modules loaded: GreetIfCool1.
Prelude> greetIfCool "downright frosty yo"
eyyyyy. What's shakin'?
Prelude> greetIfCool "please love me"
pshhhh.
```

Also note that `greetIfCool` could've been written with `cool` being a function rather than a value defined against the argument directly like so:

```
-- greetIfCool2.hs
module GreetIfCool2 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool coolness
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool v = v == "downright frosty yo"
```

4.7 Tuples

The tuple in Haskell is a type that allows you to store and pass around multiple values within a single value. Tuples have a distinctive, built-in syntax that is used at both type and term levels, and each tuple has a fixed number of constituents. We refer to tuples by how many constituents are in each tuple: the two-tuple or pair, for example, has two values inside it `(x, y)`; the three-tuple or triple has three `(x, y, z)`, and so on. This number is also known as the tuple's arity. As we will see, the values within a tuple do not have to be of the same type.

We will start by looking at the two-tuple, a tuple with two constituents. The two-tuple is expressed at both the type level and term level with the constructor `(,)`. The datatype declaration looks like this:

```
Prelude> :info (,)
data (,) a b = (,) a b
```

This is different from the `Bool` type we looked at earlier in a couple of important ways, even apart from that special type constructor syntax. The first is that it has two parameters, represented by the type variables a and b. Those have to be applied to concrete types, much as variables at the term level have to be applied to values to evaluate a function. The second major difference is that this is a *product type*, not a sum type. A product type represents a logical conjunction: you must supply *both* arguments to produce a value.

Notice that the two type variables are different, so that allows for tuples that contain values of two different types. The types are not, however, *required* to be different:

```
λ> (,) 8 10
(8,10)
λ> (,) 8 "Julie"
(8,"Julie")
λ> (,) True 'c'
(True,'c')
```

But if we try to apply it to only one argument:

```
λ> (,) 9

<interactive>:34:1:
    No instance for (Show (b0 -> (a0, b0)))
      (maybe you haven't applied enough arguments to a function?)
      arising from a use of ‘print’
    In the first argument of ‘print’, namely ‘it’
    In a stmt of an interactive GHCi command: print it
```

Well, look at that error. This is one we will explore in detail soon, but for now the important part is the part in parentheses: we haven't applied the function — in this case, the data constructor — enough arguments.

The two-tuple in Haskell has some default convenience functions for getting the first or second value. They're named `fst` and `snd`:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

As you can see from the above types, there's nothing those functions could do other than return the first or second value respectively.

Here are some examples of manipulating tuples, specifically the two-tuple:

```
Prelude> let myTup = (1 :: Integer, "blah")
Prelude> :t myTup
myTup :: (Integer, [Char])
```

```
Prelude> fst myTup
1
Prelude> snd myTup
"blah"
Prelude> import Data.Tuple
Prelude> swap myTup
("blah",1)
```

We had to import `Data.Tuple` because `swap` isn't included in the Prelude.

We can also combine tuples with other expressions:

```
Prelude> 2 + fst (1, 2)
3
Prelude> 2 + snd (1, 2)
4
```

The `(x, y)` syntax of the tuple is special. The constructors you use in the type signatures and in your code (terms) are syntactically identical even though they're different things. Sometimes that type constructor is referred to without the type variables explicitly inside of it such as `(,)`. Other times, you'll see `(a, b)` - particularly in type signatures.

You can use that syntax to *pattern match*

It's generally unwise to use tuples of an overly large size, both for efficiency and sanity reasons. Most tuples you see will be `( , , , , )` (5-tuple) or smaller.

4.8 Lists

Lists in Haskell are another type used to contain multiple values within a single value. However, they differ from tuples in three important ways: First, all constituents of a list must be of the same type. Second, Lists have their own distinct `[]` syntax. Like the tuple syntax, it is used for both the type constructor in type signatures and at the term level to express list values. Third, the number of constituents within a list can change as you operate on the list, unlike tuples where the arity is set in the type and immutable.

Here's an example for your REPL:

```
Prelude> let awesome = ["Papuchon", "curry", ":)"]
Prelude> awesome
["Papuchon","curry",":)"]

Prelude> :t awesome
awesome :: [[Char]]
```

First thing to note is that `awesome` is a list of lists of `Char` values because it is a list of strings, and a string is itself just a type *alias* for `[Char]`. This means all the functions and operations valid for lists of any value, usually expressed as `[a]`, are valid for `String` because `[Char]` is more specific than `[a]`.

```
Prelude> let alsoAwesome = ["Quake", "The Simons"]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> awesome ++ alsoAwesome
["Papuchon","curry",":)","Quake","The Simons"]

Prelude> let allAwesome = [awesome, alsoAwesome]
Prelude> allAwesome
[["Papuchon","curry",":)"],["Quake","The Simons"]]
Prelude> :t allAwesome
allAwesome :: [[[Char]]]
Prelude> :t concat
concat :: [[a]] -> [a]
Prelude> concat allAwesome
["Papuchon","curry",":)","Quake","The Simons"]
```

We'll save a full exploration of the list datatype until we get to the chapter on lists. The list data structure gets a whole chapter because lists have some interesting complexity, we're going to use them to demonstrate some things about Haskell's nonstrict evaluation, and there are *many* standard functions and constructs that can be used with lists.

4.9 Chapter Exercises

As in previous chapters, you will gain more by working out the answer before you check what GHCi tells you, but be sure to use your REPL to check your answers to the following exercises. Also, you will need to have the `awesome`, `alsoAwesome`, and `allAwesome` code from above in scope for this REPL session. For convenience of reference, here are those values again:

```
awesome = ["Papuchon", "curry", ":)"]
alsoAwesome = ["Quake", "The Simons"]
allAwesome = [awesome, alsoAwesome]
```

`length` is a function that takes a list and returns a result that tells how many items are in the list.

1. Given the definition of `length` above, what would the type signature be? How many arguments, of what type does it take? What is the type of the result it evaluates to?

2. What are the results of the following expressions?

   a) `length [1, 2, 3, 4, 5]`

   b) `length [(1, 2), (2, 3), (3, 4)]`

   c) `length allAwesome`

   d) `length (concat allAwesome)`

3. Given what we know about numeric types and the type signature of `length`, look at these two expressions. One works and one returns an error. Determine which will return an error and why.

   (n.b., you will find `Foldable t => t a` representing `[a]`, as with `concat` in the previous chapter. Again, consider `Foldable t` to represent a list here, even though list is only one of the possible types.)

   ```
   Prelude> 6 / 3
   -- and
   Prelude> 6 / length [1, 2, 3]
   ```

4. How can you fix the broken code from the preceding exercise using a different division function/operator?

5. What is the type of the expression `2 + 3 == 5`? What would we expect as a result?

6. What is the type and expected result value of the following:

```
Prelude> let x = 5
Prelude> x + 3 == 5
```

7. Below are some bits of code. Which will work? Why or why not? If they will work, what value would these reduce to?

```
Prelude> length allAwesome == 2
Prelude> length [1, 'a', 3, 'b']
Prelude> length allAwesome + length awesome
Prelude> (8 == 8) && ('b' < 'a')
Prelude> (8 == 8) && 9
```

8. Write a function that tells you whether or not a given String (or list) is a palindrome. Here you'll want to use a function called 'reverse,' a predefined function that does just what it sounds like.

```
reverse :: [a] -> [a]
reverse "blah"
"halb"
```

```
isPalindrome :: (Eq a) => [a] -> Bool
isPalindrome x = undefined
```

9. Write a function to return the absolute value of a number using if-then-else

```
myAbs :: Integer -> Integer
myAbs = undefined
```

10. Fill in the definition of the following function, using `fst` and `snd`:

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f = undefined
```

Correcting syntax

In the following examples, you'll be shown syntactically incorrect code. Type it in and try to correct it in your text editor, validating it with GHC or GHCi.

1. Here, we want a function that adds 1 to the length of a string argument and returns that result.

   ```
   x = (+)

   F xs = w 'x' 1
        where w = length xs
   ```

2. This is supposed to be the identity function, `id`.

   ```
   \ X = x
   ```

3. When fixed, this function will return 1 from the value `[1, 2, 3]`. Hint: you may need to refer back to the section about variables conventions in "Hello Haskell" to refresh your memory of this notation.

   ```
   \ x : xs -> x
   ```

4. When fixed, this function will return 1 from the value `(1, 2)`.

   ```
   f (a b) = A
   ```

Match the function names to their types

1. Which of the following types is the type of `show`?

   a) `show a => a -> String`

   b) `Show a -> a -> String`

   c) `Show a => a -> String`

2. Which of the following types is the type of `(==)`?

   a) `a -> a -> Bool`

   b) `Eq a => a -> a -> Bool`

   c) `Eq a -> a -> a -> Bool`

d) `Eq a => A -> Bool`

3. Which of the following types is the type of `fst`?

   a) `(a, b) -> a`

   b) `b -> a`

   c) `(a, b) -> b`

4. Which of the following types is the type of `(+)`?

   a) `(+) :: Num a -> a -> a -> Bool`

   b) `(+) :: Num a => a -> a -> Bool`

   c) `(+) :: num a => a -> a -> a`

   d) `(+) :: Num a => a -> a -> a`

   e) `(+) :: a -> a -> a`

4.10 Definitions

1. A *tuple* is an ordered grouping of values. In Haskell, you cannot have a tuple with only one element, but there is a zero tuple also called *unit* or `()`. The types of the elements of tuples are allowed to vary, so you can have both (String, String) or (Integer, String). Tuples in Haskell are the usual means of expressing an anonymous product.

2. A *typeclass* is a set of operations defined with respect to a polymorphic type. When a type is an instance of a typeclass, values of that type can be used in the standard operations defined for that typeclass. In Haskell, typeclasses are unique pairings of class and concrete instance. This means that if a given type a has an instance of `Eq`, it has *only* one instance of `Eq`.

3. *Data constructors* in Haskell provide a means of creating values that inhabit a given type. Data constructors in Haskell have a type and can either be constant values (nullary) or take one or more arguments just like functions. In the following example, `Cat` is a nullary data constructor for `Pet` and `Dog` is a data constructor that takes an argument:

```
-- Why name a cat? They don't answer anyway.
type Name = String

data Pet = Cat | Dog Name
```

The data constructors have the following types:

```
Prelude> :t Cat
Cat :: Pet
Prelude> :t Dog
Dog :: Name -> Pet
```

4. *Type constructors* in Haskell are *not* values and can only be used in type signatures. Just as data declarations generate data constructors to create values that inhabit that type, data declarations generate *type constructors* which can be used to denote that type. In the above example, `Pet` is the type constructor. A guideline for differentiating the two kinds of constructors is that type constructors always go to the left of the `=` in a data declaration.

5. *Data declarations* define new datatypes in Haskell. Data declarations *always* create a new type constructor, but may or *may not* create new data constructors. Data declarations are how we refer to the entire definition that begins with the `data` keyword.

6. A *type alias* is a way to refer to a type constructor or type constant by an alternate name, usually to communicate something more specific or for brevity.

   ```
   type Name = String
   -- creates a new type alias Name of the
   -- type String *not* a data declaration,
   -- just a type alias declaration
   ```

7. *Arity* is the number of arguments a function accepts. This notion is a little slippery in Haskell as, due to currying, all functions are 1-arity and we handle accepting multiple arguments by nesting functions.

8. *Polymorphism* in Haskell means being able to write code in terms of values which may be one of several, or any, type. Polymorphism in Haskell is either *parametric* or *constrained*. The identity function, `id`, is an example of a parametrically polymorphic function:

   ```
   id :: a -> a
   id x = x
   ```

   Here `id` works for a value of *any* type because it doesn't use any information specific to a given type or set of types. Whereas, the following function `isEqual`:

   ```
   isEqual :: Eq a => a -> a -> Bool
   isEqual x y = x == y
   ```

   Is polymorphic, but *constrained* or *bounded* to the set of types which have an instance of the `Eq` typeclass. The different kinds of polymorphism will be discussed in greater detail in a later chapter.

4.11 Names and variables

Names

In Haskell there are seven categories of entities that have names: functions, term-level variables, data constructors, type variables, type constructors, typeclasses, and modules. Term-level variables and data constructors exist in your terms. Term-level is where your values live and is the code that executes when your program is running. At the type-level, which is used during the static analysis & verification of your program, we have type variables, type constructors, and typeclasses. Lastly, for the purpose of organizing our code into coherent groupings across different files (more later), we have modules.

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. In small programs, they often are; in larger programs, they are often not a single letter. If there are many variables in a function or program, as is common, it is helpful to have descriptive variable names. It is often advisable in domain-specific code to to use domain-specific variable names.

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.