Chapter 1

Starting Off

We will cover a fair amount of material in this chapter and its questions, since we will need a solid base on which to build. You should read this with a computer running Haskell in front of you.

Consider first the mathematical expression $1 + 2 \times 3$. What is the result? How did you work it out? We might show the process like this:

$$\begin{aligned} & 1 + 2 \times 3 \\ \Longrightarrow \quad & 1 + 6 \\ \Longrightarrow \quad & 7 \end{aligned}$$

How did we know to multiply 2 by 3 first, instead of adding 1 and 2? How did we know when to stop? Let us underline the part of the expression which is dealt with at each step:

$$\begin{aligned} & 1 + \underline{2 \times 3} \\ \Longrightarrow \quad & \underline{1 + 6} \\ \Longrightarrow \quad & 7 \end{aligned}$$

We chose which part of the expression to deal with each time using the familiar mathematical rules. We stopped when the expression could not be processed any further.

Computer programs in Haskell are just like these expressions. In order to give you an answer, the computer needs to know all the rules you know about how to process the expression correctly. In fact, $1 + 2 \times 3$ is a valid Haskell expression as well as a valid mathematical one, but we must write `*` instead of $\times$, since there is no $\times$ key on the keyboard:

```
GHCi:
Prelude> 1 + 2 * 3
7
```

Here, `Prelude>` is Haskell prompting us to write an expression, and `1 + 2 * 3` is what we typed (the Enter key tells Haskell we have finished our expression). We'll see what `Prelude` means later. Haskell responds with the answer `7`.

Let us look at our example expression some more. There are two *operators*: $+$ and $\times$. There are three *operands*: 1, 2, and 3. When we wrote the expression down, and when we typed it into Haskell, we put spaces between the operators and operands for readability. How does Haskell process it? First, the text we wrote must be split up into its basic parts: `1`, `+`, `2`, `*`, and `3`. Haskell then looks at the order and sort of the operators and operands, and decides how to parenthesize the expression: $(1 + (2 \times 3))$. Now, processing the expression just requires doing one step at a time, until there is nothing more which can be done:

$$\begin{aligned} & (1 + \underline{(2 \times 3)}) \\ \Longrightarrow\quad & \underline{(1 + 6)} \\ \Longrightarrow\quad & 7 \end{aligned}$$

Haskell knows that $+$ refers not to 1 and 2 but to 1 and the result of 2×3, and parenthesizes the expression appropriately. We say the $\times$ operator has *higher precedence* than the $+$ operator. An *expression* is any valid Haskell program. To produce an answer, Haskell *evaluates* the expression, yielding a special sort of expression, a *value*. In our previous example, $1 + 2 \times 3$, $1 + 6$, and 7 were all expressions, but only 7 was a value. Here are some mathematical operators on numbers:

Operator	Description
a + *b*	addition
a - *b*	subtract *b* from *a*
a ∗ *b*	multiplication

The ∗ operator has higher precedence than the + and - operators. For any operator $\oplus$ above, the expression $a \oplus b \oplus c$ is equivalent to $(a \oplus b) \oplus c$ rather than $a \oplus (b \oplus c)$ (we say the operators are *left associative*). Negative numbers are written with - before them, and if we use them next to an operator we may need parentheses too:

```
GHCi:
Prelude> 5 * (-2)
-10
```

Of course, there are many more things than just numbers. Sometimes, instead of numbers, we would like to talk about truth: either something is true or it is not. For this we use *boolean values*, named after the English mathematician George Boole (1815–1864) who pioneered their use. There are just two boolean things:

```
   True
   False
```

How can we use these? One way is to use one of the *comparison operators*, which are used for comparing values to one another:

```
GHCi:
Prelude> 99 > 100
False
Prelude> 4 + 3 + 2 + 1 == 10
True
```

Here are the comparison operators:

Operator	Description
a == *b*	true if *a* and *b* are equal
a < *b*	true if *a* is less than *b*
a <= *b*	true if *a* is less than or equal to *b*
a > *b*	true if *a* is more than *b*
a >= *b*	true if *a* is more than or equal to *b*
a /= *b*	true if *a* is not equal to *b*

Notice that if we try to use operators with things for which they are not intended, Haskell will not accept the program at all:

```
GHCi:
Prelude> 1 > True

<interactive>:2:1: error:
    • No instance for (Num Bool) arising from the literal '1'
    • In the first argument of '(>)', namely '1'
      In the expression: 1 > True
      In an equation for 'it': it = 1 > True
```

Do not expect to understand the details of this error message for the moment. We shall return to them later on. You can find more information about error messages in Haskell in the appendix "Coping with Errors" on page 197.

There are two operators for combining boolean values (for instance, those resulting from using the comparison operators). The expression *a* && *b* evaluates to `True` only if expressions *a* and *b* both evaluate to `True`. The expression *a* || *b* evaluates to `True` if *a* evaluates to `True` or *b* evaluates to `True`, or both do. In each case, the expression *a* will be tested first – the second may not need to be tested at all. The && operator (pronounced "and") is of higher precedence than the || operator (pronounced "or"), so *a* && *b* || *c* is the same as (*a* && *b*) || *c*.

We shall also be using *characters*, such as 'a' or '?'. We write these in single quotation marks:

```
GHCi:
Prelude> 'c'
'c'
```

So far we have looked only at operators like +, == and && which look like familiar mathematical ones. But many constructs in programming languages look a little different. For example, to choose a course of evaluation based on some test, we use the **`if`** … **`then`** … **`else`** construct:

```
GHCi:
Prelude> if 100 > 99 then 0 else 1
0
```

The expression between **`if`** and **`then`** (in our example `100 > 99`) must evaluate to either `True` or `False`, and the expression to choose if true and the expression to choose if false must be the same sort of thing as one another – here they are both numbers. The whole expression will then evaluate to that sort of thing too, because either the **`then`** part or the **`else`** part is chosen to be the result of evaluating the whole expression:

$$\underbrace{\texttt{if}\ \overbrace{\texttt{100 > 99}}^{\text{boolean}}\ \texttt{then}\ \overbrace{\texttt{0}}^{\text{number}}\ \texttt{else}\ \overbrace{\texttt{1}}^{\text{number}}}_{\text{number}}$$

We have covered a lot in this chapter, but we need all these basic tools before we can write interesting programs. Make sure you work through the questions on paper, on the computer, or both, before moving on. Hints and answers are at the back of the book.

Questions

1. What sorts of thing do the following expressions represent and what do they evaluate to, and why?

   ```
   17
   1 + 2 * 3 + 4
   400 > 200
   1 /= 1
   True || False
   True && False
   if True then False else True
   '%'
   ```

2. These expressions are not valid Haskell. In each case, why? Can you correct them?

   ```
   1 + -1
   A == a
   false || true
   if 'A' > 'a' then True
   'a' + 'b'
   ```

3. A programmer writes `1+2 * 3+4`. What does this evaluate to? What advice would you give them?
4. Haskell has a remainder operator, which finds the remainder of dividing one number by another. It is written `` `rem` ``. Consider the evaluations of the expressions `` 1 + 2 `rem` 3 ``, `` (1 + 2) `rem` 3 ``, and `` 1 + (2 `rem` 3) ``. What can you conclude about the `+` and `` `rem` `` operators?
5. Why not just use, for example, the number `0` to represent falsity and the number `1` for truth? Why have a separate `True` and `False` at all?
6. What is the effect of the comparison operators like `<` and `>` on alphabetic characters? For example, what does `'p' < 'q'` evaluate to? What about `'A' < 'a'`? What is the effect of the comparison operators on the booleans `True` and `False`?

Chapter 2

Names and Functions

So far we have built only tiny toy programs. To build bigger ones, we need to be able to name things so as to refer to them later. We also need to write expressions whose result depends upon one or more other things. Before, if we wished to use a sub-expression twice or more in a single expression, we had to type it multiple times:

```
GHCi:
Prelude> 200 ∗ 200 ∗ 200
8000000
```

Instead, we can define our own name to stand for the expression, and then use the name as we please:

```
GHCi:
Prelude> x = 200
Prelude> x ∗ x ∗ x
8000000
```

To write this all in a single expression, we can use the **`let`** … `=` … **`in`** … construct:

```
GHCi:
Prelude> let x = 200 in x ∗ x ∗ x
8000000
Prelude> let a = 500 in (let b = a ∗ a in a + b)
250500
Prelude> let a = 500 in let b = a ∗ a in a + b
250500
```

We can use the special name `it` for the value resulting from the most recently evaluated expression, which can be useful when we forget to name something whilst experimenting:

```
GHCi:
Prelude> 200 ∗ 200
40000
Prelude> it ∗ 200
8000000
```

```
Prelude> it * 200
1600000000
```

The `it` name is not a part of the Haskell language – it is just a shortcut to make experimenting easier.

In Chapter 1, we talked about how values could be different "sorts of things" such as numbers and booleans and characters, but in fact Haskell knows about this idea – these "sorts of things" are called *types*, and every value and indeed every expression has a type. For example, the type of `False` is **Bool**. We can ask Haskell to tell us the type of a value or expression by using the `:type` command:

```
GHCi:
Prelude> :type False
False :: Bool
Prelude> :type False && True
False && True :: Bool
Prelude> :type 'x'
'x' :: Char
```

Note that commands like `:type` are not part of the Haskell language, and so cannot form part of expressions. We can read `False && True :: Bool` as "The expression `False && True` has type **Bool**". An expression always has same type as the value it will evaluate to. There is a further complication, which we shall only explain in detail later, but which we must confront on its surface now:

```
GHCi:
Prelude> :type 50
50 :: Num a => a
```

We might expect the type of `50` to be something like **Number** but it is the rather more cryptic **Num** a ⇒ a. You can read this as "if a is one of the types of number, then `50` can have type a". In Haskell, integers and other numbers are sorts of **Num**. For now, we will not worry too much about types, just making sure we can read them without being scared. The purpose is to allow, for example, the expression `50` to do the job of an integer and a real number, as and when required. For example:

```
GHCi:
Prelude> 50 + 0.6
50.6
Prelude> 50 + 6
56
```

In the first line, `50` is playing the part of a real number, not an integer, because we are adding it to another real number. In the second, it pays the part of an integer, which is why the result is `56` rather than `56.0`.

The letter a in the type is, of course, arbitrary. The types **Num** a ⇒ a and **Num** b ⇒ b and **Num** frank ⇒ frank are interchangeable. In fact, Haskell does not always use a first. On the author's machine our example reads:

```
GHCi:
Prelude> :type 50
50 :: Num p => p
```

However, we shall always use the letters a, b etc. Let us move on now to consider *functions*, whose value depends upon some input (we call this input an *argument* – we will be using the word "input" later in the book to mean something different):

```
GHCi:
Prelude> cube x = x * x * x
Prelude> cube 200
8000000
```

We chose `cube` for the name of the function and `x` for the name of its argument. If we ask for its type, Haskell will reply by telling us that its type is **Num** a ⇒ a → a. This means it is a function which takes a number as its argument, and, when given that argument, evaluates to the same sort of number. To use the function, we just write its name followed by a suitable argument. In our example, we calculated 200^3 by giving the `cube` function `200` as its argument.

The `cube` function has type **Num** a ⇒ a → a, we gave it a number `200`, and so the result is another number. Thus, the type of the expression `cube 200` is **Num** a ⇒ a (remember that the type of any expression is the type of the thing it will evaluate to, and `cube 200` evaluates to `8000000`, a number of type **Num** a ⇒ a). In diagram form:

$$\underbrace{\overbrace{\texttt{cube}}^{\textbf{Num}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{a}}\quad \overbrace{\texttt{200}}^{\textbf{Num}\ \mathsf{a} \Rightarrow \mathsf{a}}}_{\textbf{Num}\ \mathsf{a} \Rightarrow \mathsf{a}}$$

It might be easier to see what is going on if we imagine missing out the part to the left of the ⇒ symbol in each type:

$$\underbrace{\overbrace{\texttt{cube}}^{\mathsf{a} \to \mathsf{a}}\ \overbrace{\texttt{200}}^{\mathsf{a}}}_{\mathsf{a}}$$

If we try an argument of the wrong type, the program will be rejected:

```
GHCi:
Prelude>cube False

<interactive> error:
    • No instance for (Num Bool) arising from a use of 'cube'
    • In the expression: cube False
      In an equation for 'it': it = cube False
```

You can learn more about how to understand such messages in "Coping with Errors" on page 197. Here is a function which determines if a number is negative:

```
GHCi:
Prelude> neg x = if x < 0 then True else False
Prelude> neg (-30)
True
```

But, of course, this is equivalent to just writing

```
GHCi:
Prelude> neg x = x < 0
Prelude> neg (-30)
True
```

because `x < 0` will evaluate to the appropriate boolean value on its own – `True` if `x < 0` and `False` otherwise. What is the type of `neg`?

```
GHCi:
Prelude> neg x = x < 0
Prelude> :type neg
neg :: (Num a, Ord a) => a -> Bool
```

We can read this as "The argument to our function can have type `a` if `a` is an one of the class of types **Num** and also one of the class of types **Ord**. The result of the function is of type **Bool**". The class of types, or *typeclass* **Ord** is for things which can be ordered – in other words, ones on which we can use `<` and other comparison operators. A type which is one of a class of types is called an *instance* of that class. Here is another function, this time of type **Char** $\rightarrow$ **Bool**. It determines if a given character is a vowel or not:

```
GHCi:
Prelude> isVowel c = c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'
Prelude> :type isVowel
isVowel :: Char -> Bool
Prelude> isVowel 'x'
False
```

The line is getting a little long. We can type a function (or any expression) over multiple lines by preceding it with `:{` and following it with `:}`, pressing the Enter key between lines as usual. Haskell knows that we are finished when we type `:}` followed by the Enter key. Notice also that we press space a few times so that the second line appeared a little to the right of the first. This is known as *indentation*.

```
GHCi:
Prelude> :{
Prelude| isVowel c =
Prelude|   c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'
Prelude| :}
```

The start of the second line must be to the right of the name of the function: Haskell is particular about this. There can be more than one argument to a function. For example, here is a function which checks if two numbers add up to ten:

```
GHCi:
Prelude> :{
Prelude| addToTen a b =
Prelude|   a + b == 10
Prelude| :}
Prelude> addToTen 6 4
True
```

We use the function in the same way as before, but writing two numbers this time, one for each argument the function expects. The type is (**Eq** a, **Num** a) $\Rightarrow$ a $\rightarrow$ a $\rightarrow$ **Bool** because the arguments are both numbers, and both capable of being tested for equality (hence **Eq**) and the result is a boolean.

```
GHCi:
Prelude> :{
```

```
Prelude| addToTen a b =
Prelude|   a + b == 10
Prelude| :}
Prelude> :type addToTen
addToTen :: (Eq a, Num a) => a -> a -> Bool
```

Note that **Eq** and **Ord** are different. Not everything which can be tested for equality with == can be put in order with < and similar operators.

A *recursive function* function is one which uses itself. Consider calculating the factorial of a given number – the factorial of 4 (written $4!$ in mathematics), for example, is $4 \times 3 \times 2 \times 1$. Here is a recursive function to calculate the factorial. Note that it uses itself in its own definition.

```
GHCi:
Prelude> :{
Prelude| factorial n =
Prelude|   if n == 1 then 1 else n * factorial (n - 1)
Prelude| :}
Prelude> :type factorial
factorial :: (Eq a, Num a) => a -> a
Prelude> factorial 4
24
```

How does the evaluation of `factorial` 4 proceed?

$$\begin{array}{ll}
 & \underline{\texttt{factorial 4}} \\
\Longrightarrow & \texttt{4 * }\underline{\texttt{factorial (4 - 1)}} \\
\Longrightarrow & \texttt{4 * (3 * }\underline{\texttt{factorial (3 - 1)}}\texttt{)} \\
\Longrightarrow & \texttt{4 * (3 * (2 * }\underline{\texttt{factorial (2 - 1)}}\texttt{))} \\
\Longrightarrow & \texttt{4 * (3 * (}\underline{\texttt{2 * 1}}\texttt{))} \\
\Longrightarrow & \texttt{4 * (}\underline{\texttt{3 * 2}}\texttt{)} \\
\Longrightarrow & \underline{\texttt{4 * 6}} \\
\Longrightarrow & \texttt{24}
\end{array}$$

For the first three steps, the **`else`** part of the **`if`** (or *conditional expression*) is chosen, because the argument `a` is greater than one. When the argument is equal to 1, we do not use `factorial` again, but just evaluate to 1. The expression built up of all the multiplications is then evaluated until a value is reached: this is the result of the whole evaluation. It is sometimes possible for a recursive function never to finish – what if we try to evaluate `factorial (-1)`?

$$\begin{array}{ll}
 & \underline{\texttt{factorial (-1)}} \\
\Longrightarrow & \texttt{-1 * }\underline{\texttt{factorial (-1 - 1)}} \\
\Longrightarrow & \texttt{-1 * (-2 * }\underline{\texttt{factorial (-2 - 1)}}\texttt{)} \\
\Longrightarrow & \texttt{-1 * (-2 * (-3 * }\underline{\texttt{factorial (-3 - 1)}}\texttt{))} \\
\vdots & \vdots
\end{array}$$

The expression keeps expanding, and the recursion keeps going. You can interrupt this infinitely-long process by typing Ctrl-C on your keyboard (it may take a little while to work):

```
GHCi:
Prelude> factorial (-1)
^CInterrupted.
```

This is an example of a problem Haskell cannot find by merely looking at the program text – it can only be uncovered during the process of evaluation. Later in the book, we will see how to prevent people who are using our functions from making such mistakes.

One of the oldest methods for solving a problem (or *algorithm*) still in common use is Euclid's algorithm for calculating the greatest common divisor of two numbers (that is, given two positive integers a and b, finding the biggest positive integer c such that neither a/c nor b/c have a remainder). Euclid was a Greek mathematician who lived about three centuries before Christ. Euclid's algorithm is simple to write as a function with two arguments:

```
GHCi:
Prelude> :{
Prelude| gcd' a b =
Prelude|   if b == 0 then a else gcd' b (rem a b)
Prelude| :}
Prelude> gcd' 64000 3456
128
```

The function built-in function `rem` finds the remainder of dividing a by b. If we like, we can surround the function `rem` in backticks as `` `rem` `` (we have already seen this in Question 4 of the previous chapter). This allows us to put its two arguments either side, making it an operator like `+` and `||`:

```
GHCi:
Prelude> :{
Prelude| gcd' a b =
Prelude|   if b == 0 then a else gcd' b (a `rem` b)
Prelude| :}
```

Here is the evaluation:

$$\begin{array}{ll}
 & \underline{\texttt{gcd' 64000 3456}} \\
\Longrightarrow & \underline{\texttt{gcd' 3456 (64000 `rem` 3456)}} \\
\Longrightarrow & \underline{\texttt{gcd' 1792 (3456 `rem` 1792)}} \\
\Longrightarrow & \underline{\texttt{gcd' 1664 (1792 `rem` 1664)}} \\
\Longrightarrow & \underline{\texttt{gcd' 128 (1664 `rem` 128)}} \\
\Longrightarrow & \texttt{128}
\end{array}$$

Why did we call our function `gcd'` instead of `gcd`? Because Haskell has a built in function `gcd`, and we should not reuse the name. Later on, when we load our programs from files, Haskell will in fact not let us reuse the name. This is another way in which Haskell is being rather careful, to prevent us being tripped up when writing larger programs.

Finally, here is a simple function on boolean values. In the previous chapter, we looked at the && and `||` operators which are built in to Haskell. The other important boolean operator is the `not` function, which returns the boolean complement (opposite) of its argument – `True` if the argument is `False`, and vice versa. This is again built in, but it is easy enough to define ourselves, as a function of type **Bool** → **Bool**.

```
GHCi:
Prelude> :{
Prelude| not' x =
Prelude|   if x then False else True
Prelude| :}
Prelude> :type not'
not' :: Bool -> Bool
Prelude> not' True
False
```

Almost every program we write will involve functions such as these, and many larger ones too. In fact, languages like Haskell are often called *functional languages*.

A more formal look at types

Most readers will wish to skip this section, and the extra questions which relate to it, and not worry too much about types, coming back to it after a few more chapters have been worked through. However, for those who refuse to take things on trust without understanding them, it is perhaps best to tackle it now.

Every expression in Haskell has a *type*, which indicates what sort of thing it will eventually evaluate to. Simple types include **Bool** and **Char**. For example, the expression `False || True` has the type **Bool** because, when evaluated, it will result in a boolean value. So a type represents a collection of values. For example, the **Bool** type has two values: `True` and `False`, but the **Char** type has many more.

The purpose of types is to make sure that no part of the program receives something it was not expecting, and for which it cannot sensibly do anything. For example, the addition operator + being asked to add a number to a boolean. This avoids, at a stroke, a huge class of possible program misbehaviours, or bugs. Haskell can do this automatically, by working out the types of everything in the program and making sure they all fit together, and that no function can possibly receive an argument of the wrong type. This is called *type inference*, because the types are inferred (worked out) by Haskell.

When we ask Haskell what the type of 42 is, we get the surprising answer **Num** a ⇒ a, rather than something simple like **Number**. The letters a, b, c... are *type variables* standing for types. A *typeclass* like **Num** is a collection of types. So, a typeclass is a collection of types, each of which is a collection of values. A type with a ⇒ symbol in it has a left-hand and right-hand part. The left-hand part says which typeclasses one or more of the type variables on the right-hand side must belong to. So if 42 has the type **Num** a ⇒ a we may say "Given that the type variable a represents a type which is an instance of the typeclass **Num**, 42 can have type a". Remember our example where a number was used as both an integer and a real number, even though it was written the same. Of course, many types do not have a ⇒ symbol, which means either they are very specific, like **Bool**, or very generic, like a, which represents any type at all.

We have also introduced functions, which have types like $a \rightarrow b$. For example, if a is **Char** and b is **Bool**, we may have the type **Char** → **Bool**. Of course, functions may have a left-hand part too. For example, the function which adds two numbers may have the type **Num** a ⇒ a → a → a. That is to say, the function will add any two things both of a type a which is an instance of the typeclass **Num**, and the result is a number of the same type.

So this is what is rather confusing to us about the type **Num** a ⇒ a: it is actually rather harder to understand for the beginner than the function types in the previous paragraph, and yet it represents what we expect to be a simple concept: the number. All will be explained in Chapter 12.

We can have more than one constraint on a single type variable, or constraints on multiple type variables. They are each called *class constraints*, and the whole left hand part is sometimes called the *context*. For example, the type (**Num** a, **Eq** b) ⇒ a → b → a is the type of a function of two arguments, the first of which must be of some type from typeclass **Num** and the second of some type from typeclass **Eq**.

Further complicating matters, sometimes every type of a certain typeclass is by definition also part of one or more other ones. In the case of the typeclasses we have seen so far, every type in the typeclass **Ord** is also in the typeclass **Eq**. What this means is that if we list the constraint **Ord** we need not also list **Eq**.

Questions

1. Write a function which multiplies a given number by ten. What is its type?
2. Write a function which returns `True` if both of its arguments are non-zero, and `False` otherwise. What is the type of your function?
3. Write a recursive function `sum'` which, given a number n, calculates the sum $1 + 2 + 3 + \ldots + n$. What is its type?
4. Write a function `power x n` which raises `x` to the power `n`. Give its type.
5. Write a function `isConsonant` which, given a lower-case character in the range `'a'`...`'z'`, determines if it is a consonant.
6. What is the result of the expression **`let`** `x = 1` **`in let`** `x = 2` **`in`** `x + x`?
7. Can you suggest a way of preventing the non-termination of the `factorial` function in the case of a zero or negative argument?

For those who are confident with types and typeclasses. To be attempted in the first instance without the computer.

8. Here are some expressions and function definitions and some types. Pair them up.

`1`	**Ord** a ⇒ a → a → **Bool**
`1 + 2`	(**Ord** a, **Num** a) ⇒ a → a → **Bool**
`f x y = x < y`	**Num** a ⇒ a → b → c → a
`g x y = x < y + 2`	**Num** a ⇒ a
`h x y = 0`	**Num** a ⇒ b → c → a
`i x y z = x + 10`	**Num** a ⇒ a

9. Infer (work out) types for the following expressions or function definitions.

```
46 * 10        2 > 1
f x = x + x    g x y z = x + 1 < y
i a b c = b
```

10. Why are the following expressions or function definitions not accepted by Haskell?

```
True + False
6 + '6'
f x y z = (x < y) < (z + 1)
```

11. Which of the following types are equivalent to one another and which are different? Which are not valid types?

Num a ⇒ b	**Num** t1 ⇒ t1
Num b ⇒ b → a	**Num** a ⇒ a → b
(**Ord** a, **Num** a) ⇒ a → a	**Num** a ⇒ a → a
(**Num** a, **Ord** a) ⇒ a → a	**Num** a ⇒ a

12. These types are correct, but have some constraints which are not required. Remove them.

(**Eq** a, **Ord** a) ⇒ a → b → a

(**Ord** a, **Eq** a, **Eq** b) ⇒ b → b → a

Chapter 6

Functions upon Functions upon Functions

Now would be an appropriate moment to go back to Chapter 2 and look at the extra material and questions on types and typeclasses, if you have not yet done so.

Often we need to apply a function to every element of a list. For example, doubling each of the numbers in a list. We could do this with a simple recursive function, working over each element of the list:

```
doubleList :: Num a ⇒ [a] → [a]

doubleList [] = []                                   no element to process
doubleList (x:xs) = (x ∗ 2) : doubleList xs    process element, and the rest
```

For example,

$$\begin{array}{ll}
 & \underline{\texttt{doubleList [1, 2, 4]}} \\
\Longrightarrow & \texttt{1 * 2 : }\underline{\texttt{doubleList [2, 4]}} \\
\Longrightarrow & \texttt{1 * 2 : 2 * 2 : }\underline{\texttt{doubleList [4]}} \\
\Longrightarrow & \texttt{1 * 2 : 2 * 2 : 4 * 2 : }\underline{\texttt{doubleList []}} \\
\Longrightarrow & \underline{\texttt{1 * 2 : 2 * 2 : 4 * 2 : []}} \\
\overset{*}{\Longrightarrow} & \texttt{[2, 4, 8]}
\end{array}$$

The result list does not need to have the same type as the argument list. For example, we can write a function which, given a list of numbers, returns the list containing a boolean for each: `True` if the number is even, `False` if it is odd.

```
evens :: Integral a ⇒ [a] → [Bool]

evens [] = []                                          no element to process
evens (x:xs) = (x `rem` 2 == 0) : evens xs     process element, and the rest
```

For example,

$$\begin{array}{ll}
 & \underline{\texttt{evens [1, 2, 4]}} \\
\Longrightarrow & \texttt{False : }\underline{\texttt{evens [2, 4]}} \\
\Longrightarrow & \texttt{False : True : }\underline{\texttt{evens [4]}} \\
\Longrightarrow & \texttt{False : True : True : }\underline{\texttt{evens []}} \\
\Longrightarrow & \underline{\texttt{False : True : True : []}} \\
\overset{*}{\Longrightarrow} & \texttt{[False, True, True]}
\end{array}$$

It would be tedious to write a similar function each time we wanted to apply a different operation to every element of a list – can we build one which works for any operation? We will use a function as an argument:

```
map' :: (a → b) → [a] → [b]

map' f [] = []                                  no element to process
map' f (x:xs) = f x : map' f xs      process the element, and the rest
```

The `map'` function takes two arguments: a function which processes a single element, and a list. It returns a new list. We will discuss the type in a moment. For example, if we have a function `halve`:

```
halve :: Integral a ⇒ a → a

halve x = x `div` 2
```

Then we can use `map'` like this:

$$\begin{array}{ll}
 & \underline{\texttt{map' halve [10, 20, 30]}} \\
\Longrightarrow & \texttt{10 `div` 2 : }\underline{\texttt{map' halve [20, 30]}} \\
\Longrightarrow & \texttt{10 `div` 2 : 20 `div` 2 : }\underline{\texttt{map' halve [30]}} \\
\Longrightarrow & \texttt{10 `div` 2 : 20 `div` 2 : 30 `div` 2 : }\underline{\texttt{map' halve []}} \\
\Longrightarrow & \underline{\texttt{10 `div` 2 : 20 `div` 2 : 30 `div` 2 : []}} \\
\overset{*}{\Longrightarrow} & \texttt{[5, 10, 15]}
\end{array}$$

Now, let us look at that type: (a → b) → [a] → [b]. We can annotate the individual parts:

$$\overbrace{(\mathsf{a} \to \mathsf{b})}^{\text{function } \texttt{f}} \to \overbrace{[\mathsf{a}]}^{\text{argument list}} \to \overbrace{[\mathsf{b}]}^{\text{result list}}$$

We have to put the function `f` in parentheses, otherwise it would look like `map'` had four arguments. This function can have any type a → b. That is to say, it can have any argument and result types, and they do not have to be the same as each other – though they may be. The argument has type [a] because each of its elements must be an appropriate argument for `f`. In the same way, the result list has type [b] because each of its elements is a result from `f` (in our `halve` example, a and b were both numbers in typeclass **Integral**). We can rewrite our `evens` function to use `map'`:

```
isEven :: Integral a ⇒ a → Bool
evens :: Integral a ⇒ [a] → [Bool]

isEven x = x `rem` 2 == 0

evens l = map' isEven l
```

In this use of `map`, type a was **Integral** a ⇒ a, and type b was **Bool**. We can make `evens` still shorter: when we are just using a function once, we can define it directly, without naming it:

```
evens :: Integral a ⇒ [a] → [Bool]

evens l =
  map' (\x -> x `rem` 2 == 0) l
```

This is called an *anonymous function*. It is defined using `\`, a named argument, the `->` arrow and the function definition (body) itself. For example, we can write our halving function like this:

```
\x -> x `div` 2
```

and, thus, write:

```
      map' (\x -> x `div` 2) [10, 20, 30]
      ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
⟹*   [5, 10, 15]
```

We use anonymous functions when a function is only used in one place and is relatively short, to avoid defining it separately.

In the preceding chapter we wrote a sorting function and, in one of the questions, you were asked to change the function to use a different comparison operator so that the function would sort elements into reverse order. Now, we know how to write a version of the `mergeSort` function which uses any comparison function we give it. A comparison function would have type **Ord** a ⇒ a → a → **Bool**. That is, it takes two elements of the same type in typeclass **Ord**, and returns `True` if the first is "greater" than the second, for some definition of "greater" – or `False` otherwise.

So, let us alter our `merge` and `mergeSort` functions to take an extra argument – the comparison function. The result is shown in Figure 6.1. Now, if we make our own comparison operator:

```
greater :: Ord a ⇒ a → a → Bool

greater a b =
  a >= b
```

we can use it with our new version of the `mergeSort` function:

```
      mergeSort greater [5, 4, 6, 2, 1]
      ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
⟹*   [6, 5, 4, 2, 1]
```

```
merge :: (a → a → Bool) → [a] → [a] → [a]
mergeSort :: (a → a → Bool) → [a] → [a]

merge _ [] l = l
merge _ l [] = l
merge cmp (x:xs) (y:ys) =
  if cmp x y                                          use our comparison function
    then x : merge cmp xs (y : ys)                    put x first – it is "smaller"
    else y : merge cmp (x : xs) ys                    otherwise put y first

mergeSort _ [] = []
mergeSort _ [x] = [x]
mergeSort cmp l =
  merge cmp (mergeSort cmp left) (mergeSort cmp right)
    where left = take' (length' l `div` 2) l
          right = drop' (length' l `div` 2) l
```

Figure 6.1: Adding an extra argument to merge sort

In fact, we can ask Haskell to make such a function from an operator such as <= or + just by enclosing it in parentheses:

```
GHCi:
Prelude> :type (<=)
(<=) :: Ord a => a -> a -> a
Prelude> (<=) 4 5
True
```

So, for example:

$\underline{\texttt{mergeSort (<=) [5, 4, 6, 2, 1]}}$

$\overset{*}{\Longrightarrow}$ `[1, 2, 4, 5, 6]`

and

$\underline{\texttt{mergeSort (>=) [5, 4, 6, 2, 1]}}$

$\overset{*}{\Longrightarrow}$ `[6, 5, 4, 2, 1]`

Haskell provides a useful operator `.` to chain (or *compose*) functions together, so we may apply several functions to a value in order. For example, take the function `double`, we may write:

```
GHCi
Prelude> double x = x * 2
Prelude> (double . double) 6
24
```

The effect is to quadruple the number. We can write the function directly if we like:

```
GHCi
Prelude> quadruple x = (double . double) x
Prelude> quadruple 6
24
```

But, of course, we can drop the argument:

```
GHCi
Prelude> quadruple = double . double
Prelude> quadruple 6
24
```

This can lead to cleaner, easier to read programs, once you are used to it. See how we can find two ways to quadruple each element in a list using `double` and `map`:

```
GHCi:
Prelude> (map' double . map' double) [6, 4]
[24,16]
Prelude> map' (double . double) [6, 4]
[24,16]
```

The `.` or function composition operator has type $(\texttt{b} \rightarrow \texttt{c}) \rightarrow (\texttt{a} \rightarrow \texttt{b}) \rightarrow \texttt{a} \rightarrow \texttt{c}$. That is to say we give it a two functions which, when used in order, take something of type `a` to something of type `c` via something of type `b`, and something of type `a` and we get something of type `c` out. In our examples, `a`, `b`, and `c` were all numbers. Since the function composition operator is associative, that is `f . (g . h)` is equal to `(f . g) . h`, we do not need the parentheses, and may write simply `f . g . h`, for example `times8 = double . double . double`. We may define the `.` operator ourselves:

```
GHCi:
Prelude> f . g = \x -> f (g x)
```

We can, in fact, write it like this too:

```
GHCi:
Prelude> (f . g) x = f (g x)
```

See how we define an operator just like a function, since in reality it is one. As another example, consider defining +++ to mean vector addition:

```
GHCi:
Prelude> (a, b) +++ (c, d) = (a + c, b + d)
```

The techniques we have seen in this chapter are forms of *program reuse*, which is fundamental to writing manageable large programs.

Questions

1. Write a simple recursive function `calm` to replace exclamation marks in a string with periods. For example `calm "Help! Fire!"` should evaluate to `"Help. Fire."`. Now rewrite your function to use `map'` instead of recursion. What are the types of your functions?
2. Write a function `clip` which, given a number, clips it to the range $1 \ldots 10$ so that numbers bigger than 10 round down to 10, and those smaller than 1 round up to 1. Write another function `clipList` which uses this first function together with `map'` to apply this clipping to a whole list of numbers.
3. Express your function `clipList` again, this time using an anonymous function instead of `clip`.
4. Write a function `apply` which, given another function, a number of times to apply it, and an initial argument for the function, will return the cumulative effect of repeatedly applying the function. For instance, `apply f 6 4` should return `f (f (f (f (f 4)))))`. What is the type of your function?
5. Modify the insertion sort function from the preceding chapter to take a comparison function, in the same way that we modified merge sort in this chapter. What is its type?
6. Write a function `filter'` which takes a function of type a → **Bool** and an [a] and returns a list of just those elements of the argument list for which the given function returns `True`.
7. Write the function `all'` which, given a function of type a → **Bool** and an argument list of type [a] evaluates to `True` if and only if the function returns `True` for every element of the list. Give examples of its use.
8. Write a function `mapl` which maps a function of type a → b over a list of type [[a]] to produce a list of type [[b]].
9. Use the function composition operator `.` together with any other functions you like, to build a function which, given a list, returns the reverse-sorted list of all its members which are divisible by fifteen.

Chapter 3

Case by Case

In the previous chapter, we used the conditional expression **if** ... **then** ... **else** to define functions whose results depend on their arguments. For some of them we had to nest the conditional expressions one inside another. Programs like this are not terribly easy to read, and expand quickly in size and complexity as the number of cases increases.

Haskell has a nicer way of expressing choices – *pattern matching*. For example, recall our factorial function:

```
factorial :: (Eq a, Num a) ⇒ a → a

factorial n =
  if n == 1 then 1 else n ∗ factorial (n - 1)
```

We can rewrite this using pattern matching:

```
factorial :: (Eq a, Num a) ⇒ a → a

factorial 1 = 1
factorial n = n ∗ factorial (n - 1)
```

We can read this as "See if the argument matches the pattern `1`. If it does, just return `1`. If not, see if it matches the pattern `n`. If it does, the result is `n ∗ factorial (n - 1)`." Patterns like `n` are special – they match anything and give it a name. Remember our `isVowel` function from the previous chapter?

```
isVowel :: Char → Bool

isVowel c =
  c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'
```

Here is how to write it using pattern matching:

```
isVowel :: Char → Bool

isVowel 'a' = True
isVowel 'e' = True
isVowel 'i' = True
isVowel 'o' = True
isVowel 'u' = True
isVowel _ = False
```

The special pattern _ matches anything. If we miss out one or more cases – for example leaving out the final case, Haskell can warn us:

```
<interactive> warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'isVowel':
        Patterns not matched:
            p where p is not one of {'u', 'o', 'i', 'e', 'a'}
```

To enable this behaviour, you must start Haskell by writing `ghci -Wincomplete-patterns` instead of just `ghci`. Writing `ghci -Wall` enables all warnings. Haskell does not reject the program outright, because there may be legitimate reasons to miss cases out, but for now we will make sure all our pattern matches are exhaustive. Finally, let us rewrite Euclid's Algorithm from the previous chapter:

```
gcd' :: Integral a ⇒ a → a → a

gcd' a b =
  if b == 0 then a else gcd' b (a `rem` b)
```

Now in pattern matching style:

```
gcd' :: Integral a ⇒ a → a → a

gcd' a 0 = a
gcd' a b = gcd' b (a `rem` b)
```

We use pattern matching whenever it is easier to read and understand than **if** … **then** … **else** expressions.

What about this **Integral** typeclass? We did not try `:type` on the `gcd'` function in the last chapter, so we did not see this. A type of number which is an **Integral** has an additional property to one which is merely a **Num**, which is that whole-number division and remainder operations work on it. Since every thing which is an **Integral** is also a **Num**, we do not see (**Num** a, **Integral** a), but just **Integral** a in the type.

Sometimes we need more than just a pattern to decide which case to choose in a pattern match. For example, in `gcd'` above, we only needed to distinguish between `0` and any other value of `b`. Consider, though, the function to determine the sign of a number, producing -1 for all numbers less than zero, 0 for just the number zero, and 1 for all numbers above zero:

```
sign :: (Ord a, Num a, Num b) ⇒ a → b

sign x =
  if x < 0 then -1 else if x > 0 then 1 else 0
```

We cannot rewrite this using a pattern match with three cases. Haskell has a facility called *guarded equations* to help us (each line in our pattern matched functions can also be called an equation). A *guard* is an extra check to decide if a case of a pattern match is taken based upon some condition, for example `x < 0`. Here is our sign function written using guarded equations:

```
sign :: (Ord a, Num a, Num b) ⇒ a → b

sign x | x < 0     = -1
       | x > 0     = 1
       | otherwise = 0
```

There is no need to line up the equals signs vertically, but we do so to make it easier to read. The cases are considered one after another, just like when using pattern matching, and the first case which matches the guard is taken. The **`otherwise`** guard matches anything, so it comes last. We use an **`otherwise`** case to make sure every possibility is handled. We can read the `|` symbol as "when". A function can be defined using multiple equations, each of which has multiple guarded parts.

The layout rule

We have mentioned indentation, noting that Haskell is particular about it. Indeed, programs will not be accepted unless they are properly indented:

```
GHCi:
Prelude> :{
Prelude| sign x =
Prelude| if x < 0 then -1 else if x > 0 then 1 else 0
Prelude| :}

<interactive> error:
    parse error (possibly incorrect indentation or mismatched brackets)
```

Haskell is telling us that it cannot work out what we mean. Since the **`if`** … **`then`** … **`else`** … expression is part of the `sign` function, it must be indented further than the beginning of the whole `sign` expression. This applies at all times – even when the start of the whole expression is itself indented. In the case of **`if`** … **`then`** … **`else`** … itself, it is in fact permitted not to indent:

```
GHCi:
Prelude> :{
Prelude| if 1 < 0
Prelude| then 2
Prelude| else 3
Prelude| :}
3
```

However, we shall often do so, when it is easier to read:

```
GHCi:
Prelude> :{
Prelude| if 1 < 0
Prelude|   then 2
Prelude|   else 3
Prelude| :}
3
```

Consider again our `sign` function:

```
sign :: (Ord a, Num a, Num b) ⇒ a → b

sign x | x < 0     = -1
       | x > 0     = 1
       | otherwise = 0
```

We have already mentioned that lining up the equals signs is not necessary. However, we must always indent the cases. Here, we start the cases on the next line:

```
Prelude| sign x
Prelude|   | x < 0 = -1
Prelude|   | x > 0 = 1
Prelude|   | otherwise = 0
Prelude| :}
```

The layout rule is not complicated, but it can be frustrating to the beginner, especially when the error message is not clear.

Questions

1. Rewrite the `not'` function from the previous chapter in pattern matching style.
2. Use pattern matching to write a recursive function `sumMatch` which, given a positive integer n, returns the sum of all the integers from 1 to n.
3. Use pattern matching to write a function which, given two numbers x and n, computes x^n.
4. For each of the previous three questions, comment on whether you think it is easier to read the function with or without pattern matching. How might you expect this to change if the functions were much larger? Write each using guarded equations too.
5. Use guarded equations to write a function which categorises characters into three kinds: kind 0 for the lowercase letters `a`...`z`, kind 1 for the uppercase letters `a`...`z`, and kind 2 for everything else.
6. Experiment with the layout of the function definitions in this and the previous chapter. Which kinds of layout are allowed by Haskell? Which of the allowed layouts are aesthetically pleasing, or easy to read? Do any of your layouts make the program harder to change?