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:

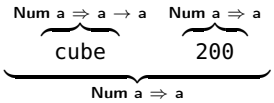
```
GHCi:
Prelude> :type 50
50 :: Num p => p
```

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):

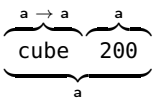
```
GHCi:
Prelude> cube x = x * x * x
Prelude> cube 200
8000000
```

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:



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:



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?

```
        factorial 4
⟹      4 * factorial (4 - 1)
⟹      4 * (3 * factorial (3 - 1))
⟹      4 * (3 * (2 * factorial (2 - 1)))
⟹      4 * (3 * (2 * 1))
⟹      4 * (3 * 2)
⟹      4 * 6
⟹      24
```

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)`?

```
        factorial (-1)
⟹      -1 * factorial (-1 - 1)
⟹      -1 * (-2 * factorial (-2 - 1))
⟹      -1 * (-2 * (-3 * factorial (-3 - 1)))
  ⋮     ⋮
```

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:

```
      gcd' 64000 3456
⟹    gcd' 3456 (64000 `rem` 3456)
⟹    gcd' 1792 (3456 `rem` 1792)
⟹    gcd' 1664 (1792 `rem` 1664)
⟹    gcd' 128 (1664 `rem` 128)
⟹    128
```

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 $a \to b$. 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