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:

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

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*.

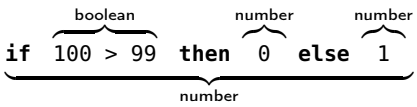
We shall also be using *characters*, such as 'a' or '?'. We write these in single quotation marks:

```
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:



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`?