Making the impossible happen

Andres Löh

Version: July 13, 2009

Contents

1 Introduction **5**

2 A tour of Haskell **7**
2.1 Lists . . . 7
2.2 A function on lists . . . 8
2.3 Evaluation . . . 8
2.4 Analysis . . . 9
2.5 The definition of "or" . . . 10
2.6 Lazy evaluation . . . 11
2.7 Type inference . . . 11
2.8 Currying . . . 12
2.9 Overloading . . . 14
2.10 Polymorphism . . . 15
2.11 Data types . . . 16
2.12 Recursion . . . 17
2.13 Higher-order functions . . . 17
2.14 Purity and effects . . . 18
2.15 The main program . . . 19

3 Getting started with GHCi **21**
3.1 Evaluating expressions . . . 21
3.2 GHCi commands . . . 23
3.2.1 Leaving GHCi . . . 23
3.2.2 Getting help . . . 23
3.2.3 Asking for the type of an expression . . . 24
3.2.4 Getting information about an identifier . . . 25
3.2.5 Loading a Haskell source file . . . 27
3.3 Making errors . . . 28
3.3.1 Lexical errors . . . 28
3.3.2 Parse errors . . . 28
3.3.3 Scope errors . . . 29
3.3.4 Type errors . . . 30
3.3.5 Type errors and type classes . . . 31

4 Basic types **35**

4.1 Booleans 35
4.2 Integers and floating point numbers 35
4.3 Lists 35
4.4 Characters and strings 35
4.5 Functions 35
4.6 Optional values and alternatives 35
4.7 Tuples 36
4.8 IO actions 36
4.9 Outlook: data structures 36

1 Introduction

This book is intended as a first course in the programming language Haskell. You will learn how to program in the language, how to use the main Haskell compiler, GHC and its interpreter effectively, how to write readable and idiomatic Haskell code and how to use the wealth of Haskell libraries that are present.

Haskell is quite a unique language. While many of the concepts present in Haskell also occur in other programming languages, it is the combination of concepts that makes Haskell particularly powerful.

Haskell programs tend to be extremely short compared to programs in other languages. This means that the same amount of information is contained in less lines of code compared to other languages. To newcomers, this can be intimidating, because sometimes, this means that a single line of code can take a long time to fully understand. With practice, however, you will learn to appreciate that things that belong together also appear in one place, rather than being distributed over many lines of code.

2 A tour of Haskell

To become familiar with Haskell, let us look at an example problem and its solution in Haskell. Given a particular number together with a list of numbers, we want to check whether the number is contained in the list.

Given the number 7 together with the list containing 6, 9 and 42, we expect the function to return "no" as an answer, but if the number was 9, we would expect the function to return "yes".

2.1 Lists

Before we can define the function, we first have to make concrete how Haskell looks at lists. Lists (as well as many other data structures) are defined *inductively*, meaning that we try to say what lists are by first describing the most simple list possible and then try to say how more complicated lists can be constructed from smaller lists.

The simplest list imaginable is the empty list. In Haskell, this list is written as $[\,]$. All lists that are not empty have at least one element. From a single element y and a list ys, we can construct a new list that is written as $\mathsf{y} : \mathsf{ys}$ (the colon : is pronounced "cons" because it constructs a list from a head element y and a tail list ys). For instance, a list with the numbers 6, 9 and 42 can be constructed stepwise:

$$6 : (9 : (42 : [\,]))$$

The empty list is extended to the left with 42, then 9, and finally 6. Note that the element added last becomes the first element of the list.

Haskell allows us to write lists in a simpler way. The above list can also be written without parentheses as

$$6 : 9 : 42 : [\,]$$

or even more conveniently as

$$[6, 9, 42]$$

All three expressions represent exactly the same list, however. The notation between square brackets is merely *syntactic sugar*, a convenient feature that does not add any extra power to the language.

2.2 A function on lists

Now, if we can build any list by starting with the empty list and adding elements to it, then we can also turn things around: if we are given a list and do not know how it looks, then there are two options: the list can be empty, or we can split off the first element. The list must thus be of the shape $[\,]$ or it must be of the shape $\mathsf{y} : \mathsf{ys}$ for some head element y and some tail list of name ys.

We can expose this distinction between two cases directly in the definition of our function, which we call elem:

$$\begin{aligned} &\mathsf{elem}\ \mathsf{x}\ [\,] &&= \mathsf{False} \\ &\mathsf{elem}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) &&= \mathsf{x} \mathbin{==} \mathsf{y} \mathbin{||} \mathsf{elem}\ \mathsf{x}\ \mathsf{ys} \end{aligned}$$

The two lines define two cases. The first line specifies that given an element x and an empty list, we return False, where False is Haskell's way of saying "no". If we have an empty list, it does not matter what x is – it can definitely not be contained in the list.

The second case is for non-empty lists. In this case, the list must be of the shape $\mathsf{y} : \mathsf{ys}$ where y is a number and ys is the tail, another list. In this case, if x is equal to y, we can certainly return True. Note that the fact that x is equal to y is expressed using a double equality operator $\mathsf{==}$, whereas the single equality operator $=$ is used to *define* an equality between a left hand side of a function and a right hand side expression.

If x and y are not equal, there is still the option that x is contained in the tail ys. This can be checked by invoking elem recursively. The operator $||$ is the logical "or". If $\mathsf{x} \mathsf{==} \mathsf{y}$, then the "or" evaluates to True without even looking at the right operand. Otherwise, it evaluates to True if and only if the recursive call evaluates to True.

2.3 Evaluation

Does the definition really work? Well, we can ask the compiler, or even more conveniently, the interpreter. However, we can also write the execution down on paper. Haskell has an extremely simple evaluation mechanism based on rewriting expressions until they can no longer be simplified. The resulting final expression is called a *value* and constitutes the result of the evaluation process.

As an example, consider the example from above. Is the number 9 contained in the list $[6, 9, 42]$. Recall that $[6, 9, 42]$ was syntactic sugar for the expression $6 : (9 : (42 : [\,]))$, so for now, we will consider the expanded version, because that makes it somewhat clearer how the definition is applied. We thus want to evaluate

$$\mathsf{elem}\ 9\ (6 : (9 : (42 : [\,])))$$

Note that in Haskell, the arguments to elem are just written next to the function call, separated by spaces. Parentheses can be used to delimit individual arguments such as the list, but are not part of the syntax of function calls.

We now look at the definition of elem. There are two cases with different left hand sides. However, only one case *matches*, namely the second. The first requires the list to be empty, but our list is not. Hence, we take the second case. Then x is bound to the first argument of elem, here 9. Furthermore, the list is split into its first element y, now 6, and the tail ys, which is $9 : 42 : [\,]$. With these *bindings*, we now substitute the left hand side of the case by the right hand side and get

$$6 == 9 \mid\mid \mathsf{elem}\ 9\ (9 : (42 : [\,]))$$

The expression now has the form c || d with c being 6 == 9 and d being the recursive call to elem. As explained earlier, in order to evaluate the "or" operator ||, we first look at the left operand. Here, we have an application of the equality test ==. This is a primitive function for numbers, and Haskell can determine that 6 == 9 is False. We thus have

$$\mathsf{False} \mid\mid \mathsf{elem}\ 9\ (9 : (42 : [\,]))$$

Now, if the left operand c of || is False, then the value of c || d is the value of the right hand side operand d. We can thus now simplify the expression to

$$\mathsf{elem}\ 9\ (9 : (42 : [\,]))$$

We are now in a situation very similar to the beginning, but with a shorter list. Again, we try to find an appropriate case in the definition of elem, and since the list is still not empty, again, the second case matches. This time, we get

$$9 == 9 \mid\mid \mathsf{elem}\ 9\ (42 : [\,])$$

As before, we look at the left operand of "or". This time, the primitive equality can determine that 9 == 9 results in True:

$$\mathsf{True} \mid\mid \mathsf{elem}\ 9\ (42 : [\,])$$

If the left operand of "or" evaluates to True, we do not need to look at the right operand and can simply return True:

$$\mathsf{True}$$

This expression True cannot be reduced any further. It is the value that results from evaluating the expression, as expected.

2.4 Analysis

Note that we can repeat the evaluation of the expression in a more formal and condensed notation that is called *equational reasoning*:

$$
\begin{aligned}
& \mathsf{elem}\ 9\ (6 : (9 : (42 : [\,])))\\
=\ & \\
& 9 == 6\ ||\ \mathsf{elem}\ 9\ (9 : (42 : [\,]))\\
=\ & \\
& \mathsf{False}\ ||\ \mathsf{elem}\ 9\ (9 : (42 : [\,]))\\
=\ & \\
& \mathsf{elem}\ 9\ (9 : (42 : [\,]))\\
=\ & \\
& \mathsf{True}\ ||\ \mathsf{elem}\ 9\ (42 : [\,])\\
=\ & \\
& \mathsf{True}
\end{aligned}
$$

In principle, any Haskell program can be evaluated like this, by rewriting it in several small steps applying the definitions of functions until it cannot be simplified any further. This is a rather simple concept of evaluation that is easy to play through for small examples in your head or on paper, and that is a strong point of Haskell. Of course, for larger programs, we will typically have many functions and at least in intermediate stages quite large expressions, so that it is not as much fun to really do it manually as for this small example. But this is what we have the compiler for, after all – to do the work automatically.

2.5 The definition of "or"

One thing that we have only informally described previously is how "or" is defined. However, "or" is not primitive. We can define it in Haskell as easily as elem itself.

Much as elem is a function on lists, "or" is a function on truth values. Recall that the key to the definition of elem was to understand how lists are defined – that they are always either of shape $[\,]$ or of shape $\mathsf{y} : \mathsf{ys}$ for a suitable element y and a tail ys. This understanding caused that we used these two possible shapes to distinguish two cases in the definition of elem.

Now, let us look at truth values. Truth values are much simpler than lists. There are exactly two of them, namely "yes", written True in Haskell, and "no" that is denoted as False. Again, we have two different options, which means that we can distinguish two different cases in the definition:

$$
\begin{aligned}
&\mathsf{True}\ ||\ \mathsf{y} = \mathsf{True}\\
&\mathsf{False}\ ||\ \mathsf{y} = \mathsf{y}
\end{aligned}
$$

If the left operand is True, then the result is True independent of the right operand y. If the left operand is False, then the resulting value is exactly the resulting value of the right operand y. If you go back to our example evaluation, you will notice that the reductions we have applied for || correspond exactly to the two cases of the definition.

2.6 Lazy evaluation

Haskell is peculiar in the sense that it only evaluates an argument to a function (or an operand of an operator) if that is actually required in order to decide which case of a definition should be applied. In the definition of "or", we see that only the left operand is analyzed, and if the left operand is True, we return True without using the right operand y again. In Haskell, this means that the right operand is never evaluated.

Of course, this behaviour is what we desire in the function elem. If we have a very long list and traverse it from the beginning, we want to stop as soon as we found the element we are looking for. Only if the element is not in the list, we really have to traverse the whole list until the end. In this case, we will ultimately arrive in the situation elem x [] which evaluates to False according to the first case in the definition of elem.

Would the case for elem x [] be omitted, Haskell could not correctly handle calls to elem where the element is not in the list, because it would not know what to do once it has traversed the list.

The use of lazy evaluation also distinguishes Haskell from many other programming languages, where arguments to functions are evaluated before the function is called. In such languages, operators like "and" and "or" are then a special case, because they implement the desired "shortcut" behaviour. In Haskell, there is no need to treat such operators in a special way, and we will see that there are actually many situations where lazy evaluation allows us to capture a concept in a simple function that would otherwise have to be built into the language.

2.7 Type inference

Haskell is a statically typed language: Every expression you write is first checked, and only if it can be assigned a valid type, the program is run. If the compiler or interpreter cannot find a valid type for a part of your program, it will complain and present you with a type error message.

At this point, you have a good reason to be confused, because the two functions we have seen so far, elem and the "or" operator ||, do not seem to contain any type declarations.

However, this absence of types does not constitute a contradiction with the fact that Haskell is statically typed. Haskell performs *type inference*. Think of type inference as a mechanical form of applying common sense. If you look at a piece of code, and you know the types of some of the expressions, you can check whether they are used consistently, and thereby also gain information about the type of the result.

For instance, if we know that 2 is a number and [] is an empty list, and that the cons-operator : takes a number and a list to a list, then we can infer that 2 : [] is a type-correct list without explicitly specifying its type. Similarly, we can detect that 2 : 3 cannot be type-correct, because the right argument of "cons" is not a list.

Haskell applies this reasoning automatically, and it is possible to write large Haskell programs without mentioning the types of functions. However, only because we can write functions without specifying their types does not mean that we should. Remember how I explained the idea of elem and ||? In both cases, I started by explaining how the type the function operates on was defined. Lists are either empty or can be seen as an application of "cons". Truth values are either True or False. From this, I concluded that we should define the function by distinguishing cases. This is not an accident, but a general principle. The type of a function gives you invaluable information about what the function is supposed to do, and can also guide you through the definition of the function. If you look at someone else's code, the types can help you to understand the definition.

Types are thus important for documentation purposes, and even though they are in most cases not needed, it is good style to put a type signature next to every function definition. For instance, the logical negation operator not can be defined as follows:

```
not True  = False
not False = True
```

This function takes a truth value as its argument, and returns a truth value. In Haskell, truth values are of type Bool, and we also call them Booleans. The type of a function that takes a Bool to a Bool is written as Bool → Bool. This is the type that Haskell infers for not, but we can also specify it explicitly, using a *type signature*:

```
not :: Bool → Bool
```

Read the double colon :: as "has type". Thus, the type signature states explicitly that not has type Bool → Bool. If a type signature is present, Haskell checks that the specified type matches the inferred type, and otherwise it complains. Therefore, type signatures also serve as an additional test that the function we defined actually has the type we had in mind.

2.8 Currying

What is the type of the "or" operator? The operator takes two arguments of type Bool and produces a Bool. Haskell provides two views on functions with multiple arguments. One option is to say that two Booleans can form a *pair* of Booleans. The type of such a pair is then written (Bool, Bool), and we can then form a function of type

```
(Bool, Bool) → Bool
```

that takes a pair of Booleans to a Boolean.

While it is possible to define functions of this type in Haskell, this is not the view that Haskell encourages, and also not the type that "or" actually has. Consider for a moment

a vending machine that offers a range of products. In order to get a product, you have to do two things: you have to throw some coins into the machine and you have to select a product using a number pad. We can model the behaviour of the machine as a function of type

$$(\mathsf{Money}, \mathsf{Number}) \to \mathsf{Product}$$

but we can also take a more sequential view. Once we throw money into the machine, the machine still expects a number. If I walked away after throwing in the money and someone else would find the machine, then to him the machine would present itself as a function that just expects a number and then yields a product, i.e., a function of type $\mathsf{Number} \to \mathsf{Product}$. Therefore, an alternative view is that the machine is a function that expects an argument of type Money and returns another function that expects an argument of type Number and produces a Product. In Haskell, this type is written as

$$\mathsf{Money} \to (\mathsf{Number} \to \mathsf{Product})$$

but we can omit the parentheses and also write

$$\mathsf{Money} \to \mathsf{Number} \to \mathsf{Product}$$

It turns out that for every function with multiple arguments we can take the two views. Either we group all arguments together in a *tuple* and provide them at once, or we provide the arguments sequentially. A function written in the second style is called *curried*, named after Haskell B. Curry.

Both styles are in principle equally expressive, but the curried style has the advantage that it lends itself better to the scenario that we "walk away from the machine" after providing some of the arguments. Curried functions can be *partially applied* to some arguments, and the remaining functions can be used later in other situations. We will see many examples of partial evaluation later.

Operators such as $||$ are seen by Haskell as binary functions in curried style, so the type of $||$ is $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$. An operator can be used as if it is a function by putting the operator in parentheses. Instead of writing $\mathsf{True} \,||\, \mathsf{False}$, we can equivalently write $(||)\ \mathsf{True}\ \mathsf{False}$. If an operator appears on its own, it also requires the surrounding parentheses, thus the type signature for "or" looks as follows:

$$(||) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$$

What about elem? The function elem was supposed to take a number and a list of numbers and produce a Boolean. The type of (integer) numbers in Haskell is Int, and the type of a list of numbers is written $[\mathsf{Int}]$. Applying what we have learnt so far, we can thus expect elem's type signature to be

$$\mathsf{elem} :: \mathsf{Int} \to [\mathsf{Int}] \to \mathsf{Bool}$$

If we specify this type signature in our program, Haskell will accept it. However, if we let Haskell infer the type for elem, we get a different type.

2.9 Overloading

Haskell infers a more general type for elem than the one we specified. So far, I have always talked about numbers when discussing elem. But take another look at the definition of elem:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

There is absolutely nothing in here that hints at numbers. Now, it could be that lists can only contain numbers in Haskell. But this is not the case. In Haskell, lists are *homogeneous*, i.e., all elements of a certain list have the same type. But apart from this restriction, the type of list elements is not further restricted. Another issue is that we test two elements for equality using the equality operator ==. Perhaps this only works for integers and has type

```
Int → Int → Bool
```

But this is too restrictive. Many (but not all) types support equality tests. Intuitively, we can compare two truth values for equality, we can compare strings for equality, and we even can compare lists for equality as long as we know when the elements are equal.

Haskell allows us to define *type classes*, i.e., collections of types that support certain functionality. One such type class is called Eq. It is the collection of types that support an equality test. The equality test operator == now expects two arguments of the *same* type, where the exact type does not matter as long as it belongs to the Eq class. The way that Haskell expresses both conditions, i.e, the same-ness of both argument types and the class membership, is by introducing a *type variable*:

```
(==) :: Eq a ⇒ a → a → Bool
```

The best way to read the double arrow ⇒ is as "if ... then ...": The operator == has the type that if a is in class Eq, it takes two arguments of type a to a Bool. The part of the type signature after the double colon, but before the double arrow is called a *class constraint*. Functions with class constraints in their type are called *overloaded*.

For elem, we now see that our initial type signature

```
elem :: Int → [Int] → Bool
```

was more specific than necessary. Instead of Int, we can use any type that supports equality. However, the type of the first argument must be the same as the type of list elements:

```
elem :: (Eq a) ⇒ a → [a] → Bool
```

This is also the type that Haskell infers. Haskell always infers the *most general* type of a function, making as few assumptions as possible.

It is allowed to give more specific types as the type signature, but once we have given the more specific type, the function also becomes restricted to that type. I.e., if we include the Int-specific type signature for elem, we cannot apply elem to characters and a list of characters afterwards, whereas with the most general type for elem we can.

Type classes are widespread in Haskell, and many functions are available on a class of types. Next to the class Eq of types that support an equality test, another quite important class is Num, the class of numeric types. In Haskell, even number literals are overloaded. The most general type of 23 is not Int, but rather Num a ⇒ a. Read this as "any type a for which Num a holds, i.e., that is a member of class Num". Of course, Int is a member of class Num, so Int is a more specific type of 23. But Float, the type of floating point numbers, also is a member of type Num, so 23 can also have type Float.

2.10 Polymorphism

What is the type of the empty list []? The type [Int] is too specific, so is [Bool]. As I have stated before, lists actually can have any element type. Since there are no elements in the empty list, every element should still be possible. We do not even need to make any assumptions about that type. It is not necessary that we can compare values of that type for equality, nor do we necessarily need any other operations available on the type of list elements. In such a case, Haskell allows us to use variables without class constraints in types: The empty list has type [a].

Another example is the operator ++ that appends to lists:

```
(++) :: [a] → [a] → [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Like elem, the "append" operator is a function on lists. So it is not surprising that we distinguish again two cases: the one where the left operand is empty, and the one where the left operand is non-empty, i.e., constructed using the "cons" operator from a head element and a tail list. If the first argument is empty, then appending is trivial: we return the second list. If the first argument is non-empty and its head element is x, then we already know the head element of the resulting list! It must also be x. We are then still left with the tail xs of the first argument, but we can recursively append it to ys.

Nothing in the definition of ++ makes any assumptions about the nature of elements in the lists. We never look at them, we do not even compare them for equality. We only need to require that both argument lists have the same element type, because we want to join them in a single list. This is expressed by the type signature, which uses a type variable for the type of list elements, and does not restrict that variable any further with a class constraint.

Functions that involve type variables without class constraints are called *polymorphic* and sometimes also *parametrically polymorphic* (because in the literature, many different

forms of polymorphism are distinguished).

2.11 Data types

While some datatypes in Haskell such as Int are built into the language, most datatypes are defined in a library, and it is easy to define your own datatypes. In fact, few other languages make it as easy to define new datatypes and encourage it as much as Haskell does. Booleans for instance can be defined by the following declaration:

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

The keyword **data** highlights that we are defining a new datatype. The left hand side, Bool, gives the name of the datatype. The right hand explains how we can construct values of type Bool. There are two alternatives, separated by a vertical bar (pronounced "or"). A value of Bool can either be True or False. The identifiers True and False are called *constructors* of the datatype Bool. As you may have noticed already, names of Haskell datatypes and constructors always start with an uppercase letter, whereas function names start with lowercase letters. Symbolic operators and constructors are an exception to this rule.

We can also define our own type of lists using **data**:

```
data List a = Nil | Cons a (List a)
```

This defines a datatype List that is parameterized by the element type a. Datatypes can thus have parameters much like functions can be parameterized. We can use the type parameter a on the right hand side of the definition. Again, there are two ways to construct a list. Either using the constructor Nil, or using the parameterized constructor Cons. In order to use Cons, we have to pass two arguments: a head element of type a and a tail list of type List a. Note that this is exactly the definition of lists that we have used before, made precise as a program. Note also that the definition of the datatype itself is recursive. Using Cons we can construct larger lists from smaller lists, and using Cons in the left hand side of a function definition, we can split non-empty lists into their head and tail.

The list consisting of 6, 9 and 42 using the above definition of List is given by

```
Cons 6 (Cons 9 (Cons 42 Nil))
```

This expression has type List Int.

Haskell's built-in lists are exactly the same as the lists we have just defined, they only use different syntax. Instead of Nil, we write [], instead of Cons, we use the operator :, written infix between its two arguments, and instead of List Int we write [Int].

2.12 Recursion

The definition of lists is recursive. The definitions of elem and ++, both functions on lists, are recursive. This is not an accident. Most data structures in Haskell, most prominently lists, are inductively defined, explaining how larger structures can be built from smaller structures. Functions on such data structures are then naturally recursive as well. As I have already hinted at, the data type we work on with a function already gives us guidelines on how to naturally define a function, and a recursive datatype naturally leads to a recursive function definition.

Recursion is ubiquitous in Haskell, and plays along well with Haskell's method of operation that works by rewriting expressions. Recursion reduces larger problems into (hopefully) smaller problems until we finally are left with a simple base case and can complete the calculation. For newcomers, recursion often requires some training. If you are coming from other programming languages, you may even have heard that recursion is bad, and that using a loop is more efficient. This is not true in Haskell. Recursion basically *is* Haskell's way of writing a loop, and Haskell's compiler is clever enough to generate as efficient code for recursion as an imperative compiler would generate for a loop whenever that is possible – recursion is actually a more general concept, and there are concepts expressible by recursion that cannot easily be programmed using loops.

2.13 Higher-order functions

However, while recursion is always present behind the scenes in Haskell, we often capture recurring patterns in their own function definitions. Compare, for instance the two functions elem and ++ on lists:

```
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys

(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

They both follow an extremely similar structure. Both have two cases, both make a distinction based on whether a list argument is constructed using [] or :. In both functions, the first case returns a value, whereas the second makes use of a recursive call on the tail of a list.

Haskell allows us to extract this pattern, and in its libraries actually contains a list traversal operator called foldr (the "r" stands for "right" – we will see later why that is). Using foldr, we can define both elem and ++ in one line:

```
elem x ys = foldr (\y r -> x == y || r) False ys
xs ++ ys  = foldr (\x r -> x : r       ) ys     xs
```

Obviously, foldr takes three arguments. Let us look at them starting at the end. The third argument specifies the list to be traversed. In the case of elem, this is the second argument of elem, and in the case of ++, it is the left operand xs.

The second argument of foldr specifies what to return if the list is empty. This is the right hand side of the first case of the original definitions, i.e., False for elem, and ys for ++.

The first argument is actually a function. In Haskell, we can define functions without giving them a name. Such functions are introduced by a backslash (that is pronounced "lambda", a nod to the lambda calculus). Between the backslash and the arrow are the arguments of the function, to the right of the arrow the result of the function. For example, the function that doubles its argument can be written as \x → 2 ∗ x or \x → x + x.

In the case of foldr, the first argument describes what to do while traversing the list if the list is not empty. In this case, we get the head element of the non-empty list, and the *result* r of the traversal on the tail, and can decide how to combine the two. Compare the definitions of the second cases of elem and ++ with the two anonymous functions and note how r appears exactly where the recursive call appeared before.

The use of recursion is not gone, but we have captured it in the definition of foldr (which is recursive), and the use of the parameter r in the anonymous function.

The fact that functions such as foldr can be defined in Haskell (this is not a built-in function) is a key to Haskell's expressive power. Functions that take other functions as parameters are called *higher-order functions*. One main application of higher-order functions is to capture commonalities in several function definitions and abstract from them.

Once you build a vocabulary of higher-order functions such as foldr and get a feeling for how they work and behave, this allows you to both write and read programs that perform rather complex tasks, but nevertheless take but a single line of code – the conciseness I mentioned at the beginning of this chapter.

2.14 Purity and effects

Haskell functions do not have so-called *side effects*. When applied to the same arguments, Haskell functions always produce the same results.

A typical example of an impure function that have side effects are a random number generator that takes a number and returns a random number between zero and the given number. Such a function cannot have type Int → Int in Haskell, because we expect it to return different random numbers even if we call it with range 22 several times. But a Haskell function of type Int → Int is guaranteed to return the same result every time we apply it to 22.

Another example for an impure function is the "function" that reads a line from the console and returns the result as a string. Such a function cannot have type String or

even $\cdots \rightarrow$ String for any choice of ... in Haskell, because the resulting String depends on what the user entered, but is not uniquely determined by the arguments of the function.

Both functions are provided with Haskell, but as I argued, they cannot have Int are String as their result types. Instead, results that may involve side effects are marked in Haskell using the built-in type former IO. A random number generator can have type Int $\rightarrow$ IO Int and a function that reads a line from the console gets type IO String.

Think of a value of type IO a as a *plan* or even a program for interaction with the outside world, that, once you're finished interacting, yields a value of type a. Reading a value from the console is such a plan and has type IO String. Given a range of type Int, to generate a random number in that range is also a plan and thus has type IO Int. And while the result of the interaction is not uniquely determined by the function inputs, the plan is actually always the same.

This ingenious indirection step makes it possible to treat effectful functions in Haskell in a pure framework. As long as we do not execute any plan, no side effects will happen. We can thus construct such plans in a pure setting and thus produce a final, large, complicated plan with many detailed steps. This plan can then be run by the Haskell runtime system, and only then, the actual side effects occur. The main advantage this view has for you as the programmer, is that you get a visual clue whether a certain operation involves interaction with the outside world of not. You can rely on the fact that a function of type Int $\rightarrow$ Int gives you the same result every time when called on the same input. For a function of type Int $\rightarrow$ IO Int, you cannot. This is convenient, because, as we will see, functions not involving IO tend to be far more flexible, and the use of effects, even in a large program, can typically be contained in a small part.

2.15 The main program

A full Haskell program consists of a possibly large number of functions. When a full program is run, then Haskell tries to evaluate a particular function called main. In other words, if you want to define a complete program in Haskell, you should define a function called main as well that drives the entire execution.

The type of main is simple enough: the main program should be able to perform interactions with the outside world, because if the main program could not, no part of the program could. After the program has run, nothing more happens, therefore no final result is produced by the "plan" that the main function describes. As we have seen before, IO is – like the type of lists – a parameterized type. What type do we choose as IO's parameter if the result is irrelevant? Haskell has a special type for this, called "unit" and denoted ( ). The type signature of main is thus always

```
main :: IO ()
```

Note that I have no problems calling main a function even though its type does not

involve a function arrow. I will generally speak of declarations that bind identifiers to expressions as *function declarations*, regardless of whether they have function type or not.

Let us end this introductory tour of the Haskell language with the program that many other tours begin with, the "Hello world" program. In Haskell, we can define the program in one line:

```
main = putStrLn "Hello world"
```

The type signature above is optional. If we don't give it, the same type is inferred for main. The function putStrLn prints a string argument on the console. It thus takes a String to an IO () plan, and consequently has type

$$\mathsf{String} \to \mathsf{IO}\ ()$$

The name putStrLn is an abbreviation for "put string line", with "line" indicating that a new line is started one the console after printing the screen.

3 Getting started with GHCi

After the introduction, it is now time to get started trying things in practice. You can only really learn Haskell if you do things. The purpose of the Haskell tour was to make sure that you have seen a few things and have already heard of some of the key ideas relevant to the Haskell language. Now, we will gradually walk through the concepts of Haskell in much more detail.

To obtain practical experience, you will need a working Haskell toolchain. In this book, I will assume that you are using the Glasgow Haskell Compiler, called GHC. Next to being an optimizing compiler, GHC comes with an amazing interactive interpreter called GHCi. Using GHCi, you can quickly load programs and then type in expressions and have them typechecked and evaluated for you on the fly. Using GHCi, you can gain a lot of experience on how Haskell works very quickly, because it is extremely easy to type in a large number of small expressions in short succession and look at their results. You can also get a good feeling for how Haskell's type inference works, because in GHCi, you can ask for the type that has been inferred for an expression.

I want to encourage you to spend a lot of time working with GHCi, and to experiment as much as you can. Ideally, you will have GHCi open on your computer while reading the book, and quickly type in things to verify that what I write does really work. I will try to explain how you can use GHCi productively, and throughout the book, I will provide many suggestions on how you can practice Haskell. In the beginning, many tasks will be possible to solve within GHCi itself. Later, you will need to write Haskell programs in an editor or development environment that you can then load into GHCi in order to play with them.

Do not limit yourself to the exercises given explicitly in this book. Set yourself your own challenges, and try out everything you can think of.

3.1 Evaluating expressions

When you start GHCi, you will see a couple of messages and then are prompted for input on a line that looks as follows:

```
Prelude⟩
```

Before the angle bracket, you see the names of Haskell modules that are currently in scope. Haskell programs are organized into several modules. When you use Haskell, you already have a large number of libraries, i.e., modules at your disposal that you

can use. In order to use a module in a program, you must explicitly import it. In order to use a module in GHCi, you have to type a command to load it.

However, there is one module, called Prelude, that is by default available in Haskell programs and also in GHCi. The prelude contains definitions of the most basic types and functions, and gives you an environment in which you can already try quite a few interesting things.

It happens that the function we have defined in the introductory tour are all defined in the prelude. So we can directly try to evaluate the example call elem 9 $[6, 9, 42]$ that we played through in the introduction. You can evaluate an expression in GHCi simply by typing it in and pressing return.

```
Prelude⟩ elem 9 [6,9,42]
True
```

For better readability, I highlight user input in blue, whereas everything the computer should output is in black. If you typed the text correctly, you get the result – in this case True printed immediately as an answer, and you are prompted for new input.

The result of the previous expression in GHCi is bound to the name it. This allows you to make further computations with the previous result without repeating the whole expression:

```
Prelude⟩ not it
False
```

Since it was bound to True, the expression not it evaluates to False. Note that now it is bound to False.

Exercise 3.1. Haskell supports arithmetic operations in a pretty standard way. You can form expressions from numbers and the operators $+$, $-$, $*$, and $/$. You can use parentheses for grouping. Perform a few calculations.

Exercise 3.2. Try the elem function on a couple of test cases and verify that it behaves as expected.

Exercise 3.3. Verify that the three expressions

$$1 : (2 : (3 : [\,]))$$
$$1 : 2 : 3 : [\,]$$
$$[1, 2, 3]$$

all lead to the same result. Note that the convenient syntax is used for printing, but the explicit form using : and $[\,]$ is still the way the value is internally represented.

Exercise 3.4. Try the append operator $(+\!\!+)$ on a few lists, for instance:

$$[\,] +\!\!+ [\,]$$
$$[1, 2, 3] +\!\!+ [\,]$$
$$[1, 2, 3] +\!\!+ [4, 5, 6]$$

Exercise 3.5. Characters are also a datatype in Haskell, denoted Char. Character literals are written in single quotes. Try:

```
'a'
' '
```

Strings in Haskell are lists of characters, written [Char] or String (both types have the same meaning). For strings, there is yet more syntactic sugar than for ordinary lists. Verify that

```
'H':'i':[]
['H','i']
"Hi"
```

all have the same result. Strings written between double quotes are called *string literals*. Always recall that they are just syntactic sugar for lists of characters. Try some familiar functions on strings, such as

```
elem 'c' "Hello"
"Hello" ++ " " ++ "world"
```

Since strings are just lists in Haskell, all functions working on lists also work on strings.

3.2 GHCi commands

GHCi can do many things more than just plainly evaluate expressions. All other functionality in GHCi is available via GHCi commands. Such commands start with a colon directly followed by the command name. GHCi commands are specific to GHCi and not a part of the Haskell language.

In the following, we will discuss the most important GHCi commands.

3.2.1 Leaving GHCi

Of course, you want to know how you can get out of GHCi again once you have had enough of Haskell (for the moment). The command that achieves this is :quit which can be abbreviated to :q:

```
Prelude⟩ :q
Leaving GHCi.
```

After printing a confirmation message, the interpreter terminates.

3.2.2 Getting help

A list of available GHCi commands can be obtained using the command :help which can be abbreviated to :h or :?. In current versions of GHCi, this list is rather long, so depending on the size of your window you may have to scroll up in order to see all of them.

3.2.3 Asking for the type of an expression

One of the most useful commands is :type. You can pass it an expression, and GHCi will show you the type that it infers for that expression.

One basic use for the command is to ask for the type of a function:

```
Prelude⟩ :t elem
elem :: (Eq a) ⇒ a → [a] → Bool
```

This is the type of the elem function as described in the introductory tour. Recall that the type says that for any type a that is a member of the Eq type class (i.e., that supports an equality test), the function requires a value of type a and then a list of type a in order to produce a Bool.

We can ask for the type of an arbitrary expression. For instance, we can partially apply the function elem to a first argument, say, the character 'x'.

```
Prelude⟩ :t elem 'x'
elem 'x' :: [Char] → Bool
```

From the result, we see that elem 'x' expects one more argument, which now has to be a list of characters, because the first argument was a single character, and then returns a Bool. If we also pass the second argument, we consequently end up with an expression of type Bool:

```
Prelude⟩ :t elem 'x' ['a','b']
elem 'x' ['a','b'] :: Bool
```

Exercise 3.6. Determine the types of the following expressions. Try to make an educated guess before looking at GHCi's reply.

```
[]
(:)
True
[True]
[[]]
["Hello","world"]
[\x → x]
```

Note from the examples that the element type of lists is really unrestricted as long as all elements of a list have the same type.

Exercise 3.7. Compare the types of

```
[]
""
```

Exercise 3.8. Determine the types of the following expressions. Try to make an educated guess before looking at GHCi's reply.

```
\x → x
\x → x == x
\x y → x == y
```

Exercise 3.9. Ask for the type of a number literal.

```
0
```

Recall the explanation from Section 2.9 on how number literals in Haskell are overloaded. Depending on the context, 0 can stand for an element of type Int or an element of type Float. There are a number of type classes in Haskell categorizing different number types: Num, Integral, Fractional, RealFrac. We will discuss all these classes later.

Determine the types and results of the following expressions:

```
2.5
1 + 2.5
1 + 2
mod 7 3
floor 3.5
[1, 2, 3]
[1, 1.5, 2]
```

3.2.4 Getting information about an identifier

A command that is somewhat related to :type is :info which can be abbreviated to :i. Where :type takes a Haskell expression as an argument, the command :info only takes a single identifier.

Depending on what the identifier represents, different sorts of information is displayed. For functions, the type is displayed, much as it would be with :type. But in addition, you may learn more, such as the module in which the identifier has been bound to its current meaning. Here is an example:

```
Prelude⟩ :i (++)
(++) :: [a] → [a] → [a]   -- Defined in GHC.Base
infixr 5 ++
```

The type of (++) is printed, plus the module it is defined in. This turns out not to be Prelude, but GHC.Base. GHCi is exporting an implementation detail here, as other Haskell implementations will not have a module called GHC.Base. GHC's version of Prelude imports functionality from GHC.Base and then reexports it. Information about the module where a function is actually defined is useful if you want to look at the actual code that defines a function. It can also be used by development environments to jump to the sources of that module and show you the code.

The line **infixr** 5++ says something about how ++ can be used as an infix operator. The 5 indicates how strongly it binds to its operands compared to other operators, and the r of **infixr** indicates that it is right-associative, meaning that x ++ y ++ z means the same as x ++ (y ++ z) rather than (x ++ y) ++ z. We will discuss **infix** statements in much more detail later.

For other identifiers, we get other information:

```
Prelude⟩ :i False
data Bool = False | ...    -- Defined in GHC.Bool
```

The ... is not valid Haskell syntax, but what the answer is supposed to show is that False is a constructor of the datatype Bool.

If we ask for information on a datatype such as Char, we get the following:

```
Prelude⟩ :i Char
data Char = GHC.Types.C # GHC.Prim.Char #    -- Defined in GHC.Types
instance Bounded Char    -- Defined in GHC.Enum
instance Enum Char    -- Defined in GHC.Enum
instance Eq Char    -- Defined in GHC.Base
instance Ord Char    -- Defined in GHC.Base
instance Read Char    -- Defined in GHC.Read
instance Show Char    -- Defined in GHC.Show
```

This shows you the definition of Char. Again, this is an implementation detail of GHC. Other Haskell implementations may choose to implement Char in some other way. For now, rather think of Char as an abstract type containing character literals as values. We will discuss so-called *unboxed types*, as indicated by the use of the #, later.

More importantly, asking information of a datatype also shows you the type classes it belongs to. A line such as

```
instance Eq Char
```

means that the datatype Char is a member of the equality class, i.e., that it supports an equality test.

Exercise 3.10. Ask for information about a couple of identifiers you have already encountered. Never mind if you do not understand all of the information that is presented yet – we will discuss most of it later. Here are a few examples you could try:

```
Bool
String
Eq
```

3.2.5 Loading a Haskell source file

You can load a Haskell source file into the interpreter with using :load, abbreviated as :l. You pass it the name of a file (you can give a path to a file in the syntax of the operating system you use, too). The file is then loaded.

For instance, if we open an editor and enter the following program

```
welcome :: IO ()
welcome = putStrLn "Welcome to Haskell!"
```

then save the file under the name `Welcome.hs`, we can load it as follows:

```
Prelude⟩ :l Welcome.hs
[1 of 1] Compiling Main      ( Welcome.hs, interpreted )
Ok, modules loaded: Main.
*Main⟩
```

The messages state that the module contained in the Haskell file is loaded. Haskell allows us to choose a name for a module – we have not discussed that yet. As long as we do not specify a module name explicitly, Haskell by default assigns the name Main to a module. Since we just load the file into the interpreter, GHCi lists that the module is interpreted. We can load compile files using GHC and then load them into the interpreter as well. They will be listed as "compiled", and code contained in them will be executed slightly faster. For our current purposes, interpretation is good enough.

Finally, the prompt changes to *Main, because Main functions from Main are now available for use in the interpreter. For the moment, think of the star as an indication that the module is interpreted. We will discuss its exact meaning once we know more about the module system.

We can now call the function welcome:

```
Prelude⟩ welcome
Welcome to Haskell!
```

You can reload the current module (for instance, if you have modified the source file) using the command :reload or :r without a parameter.

Exercise 3.11. Create a file `Welcome.hs` as indicated above, load it into the interpreter, and evaluate the expression welcome.

Then change the file by adding a definition of

```
name :: String
```

Define name to be a string literal containing your name. Now adapt welcome to be

```
welcome = "Welcome to Haskell, " ++ name ++ "!"
```

After you made the modifications, reload the file into the interpreter and run welcome again.

3.3 Making errors

It is important to learn how to deal with errors. Errors belong to programming, and even though you are of course encouraged to think before you type, it is natural to make errors, and one should not be afraid to make them. I will try to explain what kinds of errors you are likely to get from GHC and GHCi and how to deal with them.

Note that the exact error messages you get are extremely compiler-specific and may change from version to version of GHC. The book is based on GHC version 6.10.3, so you may get different messages. Despite the fact, the reasons for the errors are the same, so you will hopefully be able to understand the explanations.

The most likely error messages that you may have encountered already while trying to solve the exercises above are lexical errors, parse errors, scope errors and type errors. These are layered in the sense that if you get a parse error, you know that there are no lexical errors in your input. If you get a scope error or a type error, you know that are are neither parse nor lexical errors. GHC currently stops processing your input once it finds a lexical or a parse error, but you may get multiple scope and type errors reported at the same time.

3.3.1 Lexical errors

The difference between lexical errors and parse errors is an implementation detail that GHCi exposes to you. Lexical errors are very fundamental syntactic errors that are easy to localize and fix. The most common case is that you write an ill-formed literal. For instance, if you forget the end quote of a character literal, you get

```
Prelude⟩ 'c
⟨interactive⟩ : 1 : 2 :
  lexical error in string/character literal at end of input
```

Every error message starts with a location. If an error occurs in the interactive input, the text "interactive" indicates that. Otherwise, the file name of the error is given there. After the file name follows the line and column number of the input. In a one-line interactive input, the line number is not very informative, but the column number may be.

3.3.2 Parse errors

Most syntax errors are parse errors. In the beginning, you are likely to get many parse errors, because you are not yet familiar with the peculiarities of the language's syntax. Over time, this will be better. Parse errors can have a wide range of causes, such as incomplete syntactic constructs, the use of certain constructions in the wrong places, unbalanced parentheses etc. The best approach to solving a syntax error if you do not immediately spot it is to look carefully back at the explanation of the constructions you

are using and try to verify that you have understood everything. If that does not work, try to simplify the expression and check whether parts of it are syntactically correct.

In the following, we discuss a few examples of parse errors.

```
Prelude⟩ + 2 3
⟨interactive⟩ : 1 : 0 : parse error on input '+'
```

In this case, you have tried to use an operator as if it is a function. To do so, you have to put it in parentheses and write $(+)\ 2\ 3$. Alternatively, you can use the operator in infix position and write $2 + 3$.

```
Prelude⟩ 10 − (2 + 3 − (1 + 1)
⟨interactive⟩ : 1 : 21 : parse error (possibly incorrect indentation)
```

Forgotten parentheses lead to parse errors. Note that while a parse error resulting from an extra closing parenthesis typically points to the offending position, the error for a forgotten closing parenthesis can point at the wrong position. In the example above, the error points at the end of the input, but perhaps we intended the closing parenthesis to be placed after the 3.

The error message also illustrates another point. Sometimes, GHC tries to guess what is wrong, and often, such guesses and hints are helpful. But it is important to keep in mind that they are just that: guesses. GHC is not always right when it suggests something, and in this case, possibly incorrect indentation has nothing to do with the cause of the error.

```
Prelude⟩ [7; 9]
⟨interactive⟩ : 1 : 2 : parse error on input ';'
```

This probably is an attempt to write a list with elements 7 and 9. In the convenient notation, list elements are separated by commas, not by semicolons, hence the error.

3.3.3 Scope errors

A scope error occurs when you refer to a name that is not defined in the current situation. Typically, this is an indication of a typo and can easily be fixed by correcting the name. Sometimes, it may indicate that you actually forgot to make a definition, and another possible explanation is that you actually meant to write a character or string literal instead.

```
Prelude⟩ elim 3 []
⟨interactive⟩ : 1 : 0 : Not in scope: 'elim'
```

This expression refers to a function `elim`, but unlike `elem`, `elim` is not defined in the prelude.

3.3.4 Type errors

One you get more familiar with Haskell, the highest percentage of compiler errors you will get will be type errors. Type errors are reported for programs that are syntactically correct, but cannot be assigned a valid type. In most cases, this points to a logical error in your program, and when running the program, it might crash or otherwise misbehave.

The advantage of a type error compared to a crash while running the program is that you find the error during the compilation of the program, independent of a particular program run and without any testing. A strong type system therefore prevents a significant number of errors and thereby reduce, but certainly not eliminate the need for testing.

The disadvantage of type errors is that you have to understand the errors and fix the before you are allowed to run the program. Some programmers who are experiences with programming languages that have weaker type systems are used to debugging programs while running them. Working with Haskell may require you to develop a new skill of locating the true causes of type errors. This skill can definitely be trained, but it requires practice.

Type errors can be very diverse, and it would be unwise to discuss all forms of type errors now, when we have not yet looked at the language in real detail. I will point to potential type errors throughout the book in subsequent exercises. For now, we will only look at a few small examples.

A classic example of a type error is when you pass an argument of the wrong type to a function:

```
Prelude⟩ not 'c'
⟨interactive⟩ : 1 : 4 :
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the first argument of 'not', namely 'c'
  In the expression: not 'c'
  In the definition of 'it': it = not 'c'
```

The function not is of type Bool → Bool, but we passed 'c', which is a Char. The argument position is expected to have type Bool, but 'c' has been inferred to have type Char. Hence the error.

The subsequent lines indicate the position in the error, starting very precisely, then becoming more general. The reason for this hierarchy of positional messages is that sometimes, the compiler is not able to produce the correct error location in terms of a line and column number, a shortcoming of the current GHC implementation.

Anyway, the relevant positional information is stated after the nature of the error. The first argument of function not is to blame.

Note that the final line mentions the definition of it. This is related to the explanation of it in Section 3.1. Whenever you evaluate an expression interactively in GHCi, it is

implicitly treated as a binding of the identifier it.

Another possibility of a type error is to misremember the number of arguments a function expects. Here is what happens if we accidentally apply not to two arguments:

```
Prelude⟩ not True 'c'
⟨interactive⟩ : 1 : 0 :
  Couldn't match expected type 'Char → t'
        against inferred type 'Bool'
  In the expression: not True 'c'
  In the definition of 'it': it = not True 'c'
```

The problem here is that not True is a valid function application, so the subexpression not True is inferred to have type Bool. However, we are applying this expression of type Bool to a Char. For this to work, not True would have to be a function from type Char to any type t, i.e., not True is expected to have type Char → t. This is the mismatch that is reported.

Exercise 3.12. Type errors often arise if you forget parentheses. Try to evaluate the expression

```
not not True
```

This is interpreted as

```
(not not) True
```

whereas

```
not (not True)
```

is correct and of type Bool. Try to understand the type error you get from the first two expressions. Is the error an instance of passing an argument of the wrong type, or of passing too many arguments?

3.3.5 Type errors and type classes

Forgetting arguments to a function in GHCi can lead to a rather strange type error:

```
Prelude⟩ not
⟨interactive⟩ : 1 : 0 :
  No instance for (Show (Bool → Bool))
    arising from a use of 'print' at ⟨interactive⟩ : 1 : 0 − −2
  Possible fix: add an instance declaration for (Show (Bool → Bool))
  In a statement of a 'do' expression: print it
```

Note that not is a perfectly type-correct expression. Asking for its type succeeds:

```
Prelude⟩ :t not
not :: Bool → Bool
```

The expression we try to execute is a function, and the problem here results from the fact that GHCi does not know how to print a function to the screen. How values are shown by GHCi is governed by a type class called Show (recall the introductory explanation of type classes from Section 2.9). Functions in general, and functions of type Bool → Bool in particular, are not a member of this type class Show. This is the error.

The explanation further mentions the function print. The function

```
print :: Show a ⇒ a → IO ()
```

is implicitly called by GHCi in order to print the result. It is much like the function

```
putStrLn :: String → IO ()
```

that we have encountered in the "Hello world" program (Section 2.15), but where putStrLn works only on strings, the function print works on all types that are in Show.

The suggested fix is to make Bool → Bool a member of Show. While this would prevent the error, there is no good way to print arbitrary functions, so the suggestion is not really practical.

Finally, the error message mentions a **do** expression. This is another implementation detail not actually relevant to the error. We explain **do** expressions once we treat IO in more detail, and again later.

Type classes can cause other error messages that may be surprising at first sight. If we change the type error example not 'c' to not 0, we get:

```
Prelude⟩ not 0
⟨interactive⟩ : 1 : 4 :
  No instance for (Num Bool)
    arising from the literal '0' at ⟨interactive⟩ : 1 : 4
  Possible fix: add an instance declaration for (Num Bool)
  In the first argument of 'not', namely '0'
  In the expression: 'not 0'
  In the definition of 'it': it = not 0
```

As discussed in Section 2.9 and Exercise 3.9, number literals are overloaded in Haskell and can actually assume any type that is a member of the Num class. Since not expects a Bool as its argument, the expression num 0 would actually be correct if Bool was a member of the Num class. But in Haskell, Bool is not a number type, hence the error. In some other languages, there are automatic coercions that equate numbers with Booleans, such as 0 being False and 1 being True. Haskell does not apply such automatic coercions. If you want to perform such a transformation, you have to call a function to

perform it. (Such functions exist – they are called fromEnum and toEnum and explained later.)

Let us conclude our initial exposition of type errors at this point. As I promised earlier, we will return to this topic regularly, and discuss type errors specific to certain language constructs at the appropriate moments.

Exercise 3.13. Not all types are in the Eq type class. Functions, for instance, do not support an equality test. Try to evaluate

```
elem == elem
```

and try to understand the type error message.

Exercise 3.14. Sometimes, an expression that looks incorrect actually turns out to be correct: try to ask for the type (using :t) of

```
elem "Hello"
```

Recall from Exercise 3.6 that there is no problem at all to form list of lists, and from Exercise 3.5 that String is a synonym for [Char]. Try to add a second argument to the above call of elem such that the call is type correct and evaluates to True.

4 Basic types

In this chapter, we will look at the types that are most fundamental in the Haskell language. Many of these types we have already encountered in the introduction.

4.1 Booleans

Booleans are one of the simplest types. Booleans can be defined using the declaration

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

There are two ways to construct booleans: the constructor True and the constructor False, and both do not take any arguments.

if-then-else

4.2 Integers and floating point numbers

Int, Integer, Float, Double

Natural numbers.

4.3 Lists

```
data [a] = [] | a : [a]
```

4.4 Characters and strings

4.5 Functions

Lambda abstraction.

4.6 Optional values and alternatives

```
data Maybe a = Nothing | Just a
```

```
data Either a b = Left a | Right b
```

4.7 Tuples

4.8 IO actions

4.9 Outlook: data structures

Finite maps. Sequences.