The first 111 steps with Haskell and GHCi

Andres Löh
Well-Typed LLP

April 23, 2012

This introduction takes you slowly through your first steps with Haskell. Most of the exercises are easy, some may even sound too easy, but try them nevertheless. They're intended to give you more familiarity with the interpreter, and also to let you encounter typical error situations in a harmless situation, so that you can learn to deal with them.

As a guideline, you should not have to spend more than a minute on most of the exercises. If you cannot solve an exercise after a couple of minutes, don't spend too much time on it. Rather go on and try again later.

A few exercises are marked explicitly as being difficult. It is not necessary to solve them on a first pass, and they may take longer than 5 to 10 minutes. They are included as a challenge.

1. If you haven't already, install the *Haskell Platform*. The Haskell Platform contains GHC, a Haskell compiler, an interpreter GHCi, plus some very frequently used libraries and tools. In particular, it contains the `cabal` program that you can use to install additional libraries on your system later.

For more information on how to get the platform, refer to

```
http://hackage.haskell.org/platform/
```

If you want to test that everything works as expected, you can do "Haskell in 5 Steps":

```
http://haskell.org/haskellwiki/Haskell_in_5_steps
```

2. Start up GHCi. Depending on your environment, this can either be done by choosing GHCi from a menu, by clicking on an icon, or by opening a command line and at the command line prompt `$`, typing

```
$ ghci
```

You're confronted with a welcome message and finally, the GHCi prompt, which will usually say `Prelude>`, but we'll abbreviate it here simply to `>`.

Getting started with GHCi

3. Type

```
> :?
```

to get help. This lists all the *commands* available in the *interpreter* (i.e., in GHCi). Commands all start with a colon `:`. Commands are not Haskell code, they just tell the interpreter what to do. As you can see, there are a lot of commands, but we'll only need a few of them.

4. Type

```
> :q
```

(or `:quit`) to leave the interpreter again. This is a useful command to know if you ever get (temporarily) tired of practicing Haskell.

5. Start GHCi again.

```
$ ghci
```

Now you know how to start and leave GHCi.

Basic arithmetic

6. Type

```
> 2
```

Generally, you can type *expressions* at the prompt. The expressions are evaluated, and the final value is printed. For `2`, there's not much to evaluate, and the final value is the same as what you typed.

7. Type

```
> 2 + 2
```

There are binary *operators* in Haskell that are written in usual infix notation.

8. Type

```
> (+) 2 2
```

All binary operators can also be written in prefix notation. The operator is then surrounded by parentheses. Arguments are just separated by spaces, no parentheses or commas.

9. A special feature of the GHCi interpreter is that the last evaluated value is always available for further computation under the name `it`:

```
> it
```

Also try:

```
> it + 2
> it + 3
> it + it
```

10. Operators have their standard priorities. In particular, `(*)` binds stronger than `(+)` or `(-)`. Type

```
> 6 * 11 - 2
```

11. Parentheses can be used to direct the computation. Type

```
> 6 * (11 - 2)
```

and compare the result here with the result from Exercise 10.

12. Try

```
> (-) ((*) 6 11) 2
> (*) 6 ((-) 11 2)
```

The same as the two expressions before, but in prefix notation – just to show that it is possible.

13. Type

```
> 6 * (11 - 2
```

If you forget parentheses or other make syntactic mistakes, you'll often get a *parse error*. This indicates that your expression isn't legal Haskell. You can press the "up arrow" key on your keyboard to get back the expression you typed, correct it, and try again.

14. Try

```
> True
> "Hello"
```

Not only numbers can be evaluated. Haskell has several types, for instance Boolean values and strings.

Booleans

15. Type

```
> True || False
```

There are special operators on Boolean values. The `(||)` operator is the logical "or". It returns true if at least one of the arguments is true.

16. Try

```
> not True
> not False
```

Here, `not` is a (predefined) function that negates a Boolean value. The argument of `not` is just separated with a space, no parentheses are necessary. Although

```
> not (True)
```

would work, this syntax is not typically used by Haskell programmers.

17. Of course, the logical "and" exists as well. Try

```
True && True
False && True
(not False || True) && (False || True)
```

What's the answer?

18. Type

```
> true
```

Haskell is case-sensitive, i.e., it matters if you use lower- or upper-case characters. In Haskell, identifiers that start with a lower-case letter are abbreviations for expressions (i.e., predefined functions such as `not`), whereas identifiers that start with an upper-case letter are somewhat special – so-called *data constructors*, such as `True` and `False`.

If you type an identifier that hasn't been defined yet, you'll get an error message saying that the identifier is "not in scope." Scope refers to the area of a program where an identifier is known, so "not in scope" indicates that the identifier is unknown at this point.

19. Try

```
> True || False && True
> not False && True
```

What does this tell you about the priorities of the operators `(||)` and `(&&)`? Also, what does it tell you about the priority of function application? How would you have to place parentheses in each of the two examples to get the opposite result?

20. In these exercises, we are using lots of predefined functions. Most of these functions are not actually *primitive*, but simply defined as Haskell programs in a library called `Prelude` that is automatically preloaded. At a later stage, you will learn about how to manage Haskell modules, i.e., how to load functions from other libraries, or how to hide functions from the `Prelude` that you don't actually want to use. For now, try

```
> :bro
```

This will show a long list of all the functions that are currently available to you, and their type signatures. We will learn more about type signatures later. See if you can find `not` in the list.

Strings

21. For strings, there are also operators. For instance, `(++)` concatenates two strings – try:

```
> "Hello" ++ " " ++ "world"
```

22. The function `length` computes the length of a string – try:

```
> length "Hello"
> length "world"
> length ""
```

23. Try also:

```
> head "Hello"
> tail "Hello"
> last "Hello"
> init "Hello"
> reverse "Hello"
> null "Hello"
```

What do these functions do? Gain confidence in your assumptions by trying more examples.

24. If you type

```
> head ""
```

you get an *exception*. As you probably have discovered in Exercise 23, `head` tries to determine the first element of a string. On an empty string, it fails! Exceptions are different from all the errors you have seen so far, because they occur while the program is executed. In contrast, parse errors (Exercise 13) and scoping errors (Exercise 18) occur *before* the program is executed. This difference is more pronounced if you develop real programs and compile them. Then parse and scope errors occur during compilation, whereas an exception such as caused by `head` will only occur at runtime. We also say that exceptions are *dynamic* errors, whereas the others are *static* errors.

25. Which other of the functions in Exercise 23 cause an exception when called on the empty string?

26. Type

```
> "Hello
```

If you forget quotes around a string and in a few other situations, you'll get a "lexical error". Again, this indicates that you haven't provided a legal Haskell expression, and given the distinction made in Exercise 24, it is a static error. Lexical errors are much like parse errors, but on an even more fundamental level.

27 (medium, recommended)**.** Type

```
> not "Hello"
```

If you try to use logical negation on a string, you get yet another sort of error. This is – once you get used to programming in Haskell – the most frequent kind of error you'll be confronted with: *a type error*. Type errors usually indicate that something is semantically wrong with your program. Type errors come in many flavours, and require a lot of practice to read and understand. Therefore it is actually good to make type errors, because it gives you practice understanding them!

Here, you'll get the following message:

```
Couldn't match expected type ‘Bool’ with actual type ‘[Char]’
In the first argument of ‘not’, namely ‘"Hello"’
In the expression: not "Hello"
In an equation for ‘it’: it = not "Hello"
```

The first line tells you *what* went wrong, the other lines tell you more about *where* it went wrong.

The first line says that a `Bool` was expected where a `[Char]` was given. Now, as we'll see soon, `[Char]` just means a string. So, a Boolean was expected where a string was given. This makes sense: logical negation via `not` expects as its argument a Boolean value, but we've passed `"Hello"`, which is a string. Indeed, this is what the second line says: the argument of `not`, namely `"Hello"` is blamed. The other lines give more information about the context and are not so important (note, however that the last line mentions `it`, the identifier that's always implicitly bound to the last result in the interpreter, see Exercise 9).

Note that type errors are *static errors*, too. Haskell is a statically typed language, and type errors will be detected and reported *before* a program is executed.

Types

28. Every Haskell expression has a type, and there is an interpreter command to ask for that type.

```
> :t True
True :: Bool
```

The symbol `::` reads "has the type", so the answer is that `True` has the type `Bool` of Boolean values.

29. Let's try a string next:

```
> :t "Hello"
"Hello" :: [Char]
```

This time, the answer is that `"Hello"` has the type `[Char]`. This in turn stands for a "list with elements of type `Char`," where `Char` is the type of single characters. So in Haskell, strings are just lists of single characters, and the notation `"Hello"` is actually just an abbreviation for a notation that makes the list-like character much more obvious – try:

```
> ['H','e','l','l','o']
> :t ['H','e','l','l','o']
> 'H'
> :t 'H'
```

Single characters are written in single quotes, strings are lists of characters, but can be written shorter between double quotes.

30. Recall function `head` from Exercise 23. Try again

```
> head "Hello"
```

Does it result in a string or in a single character? Verify your result using

```
> :t head "Hello"
```

Note that you can apply `:t` to arbitrary expressions, not just values. How about the expression `tail "Hello"` – does it return a string or a single character?

31. Functions (and operators) also have types:

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

The type `Bool -> Bool` is the type of functions that expect a `Bool` as parameter and deliver a `Bool` as result. Now look again at Exercise 27 with the type error. There, we tried

```
not "Hello"
```

The function `not` expects a `Bool`, and `"Hello"` is a `[Char]`, hence the error.

Lists

32 (medium)**.** Let's try one of our string functions next:

```
> :t length
length :: [a] -> Int
```

The result is somewhat surprising. The function returns an integer (i.e., an `Int`), ok. But it doesn't take a string, i.e., a `[Char]`, but instead a `[a]`? What does the `a` mean?

It means that we don't care! The `a` is a *type variable*. A type variable is a bit like a joker – we can choose any type to take `a`'s place! So `length` computes the length of *any* list – not just lists of characters, but also lists of numbers, or even lists of lists. Try to guess the answers before trying the expressions in the interpreter:

```
> length [1,2,3]
> length [[1,2],[1,2,3],[],[99]]
> length ["Hello","world"]
```

If we compute the length of a list of lists, the length of the inner lists is irrelevant.

Types like the type of `length`, which contain type variables, are called *polymorphic*, because it is like they have many different types at once.

33. We can be even more adventurous. In Haskell, functions are just values like anything else. So, we can put functions into lists!

```
> length [length,head]
```

Here we have a list of two functions.

34. Recall function `head` from Exercise 23, 30 and 33. Try to guess what the type of `head` is. Verify it in the interpreter.

35. Guess and check the types of `tail`, `reverse` and `null`.

36. Guess and check the type of `[True,False,False]`.

37. All elements of a list must be of the same type! Let's try what happens if this is not the case:

```
> [True,"Hello"]
```

A type error again! And very similar to the one before:

```
Couldn't match expected type ‘Bool’ with actual type ‘[Char]’
In the expression: "Hello"
In the expression: [True, "Hello"]
In an equation for ‘it’: it = [True, "Hello"]
```

Again, we have provided a string where a Boolean value was expected. Again, `"Hello"` is blamed. This time, the first element of the list was a `Bool` (namely `True`), so the type checker inferred that we're writing a list of Booleans.

38. Guess what the type of the empty list is! Think about this first. Only then try it in the interpreter.

39. Let us produce another type error.

```
> [[False],True]
```

Here, we get:

```
Couldn't match expected type `[Bool]' with actual type `Bool'
In the expression: True
In the expression: [[False], True]
In an equation for `it': it = [[False], True]
```

Because the first element is of type `[Bool]`, and the second of type `Bool`.

40 (difficult)**.** The alert reader might have discovered an apparent inconsistency in what we've discussed so far. I said: the elements of lists all have to be of the same type. We have seen in the previous exercises that `length` and `head` are of different types. Try again:

```
> :t length
> :t head
```

But we have successfully computed

```
> length [length,head]
```

in Exercise 33. Can you guess why?

Try the following:

```
> :t []
> :t [length]
> :t [head]
> :t [length,head]
```

Can you explain these types?

41. Try to guess what

```
> (head [length]) "Hello"
```

does, and then verify your guess in the interpreter. Recall that functions are just ordinary values in Haskell! They can be passed around, put into data structures such as lists, and be arguments and results of other functions. Even though it might seem unusual, don't let it confuse you.

42 (difficult)**.** Try to guess what

```
> (head [length,head]) "Hello"
```

does, and then verify your guess in the interpreter. Try to explain! What does this say about Haskell's type system?

Tuples

43. Every element in a list has the same type, but lists can contain arbitrarily many elements. Haskell also provides *tuples*. Tuples have a fixed length, but elements of different types can be combined. Try the following expressions:

```
> (1,2)
> (1,"Hello")
> (True,id,[1,2])
> (1,2,3)
> (1,(2,3))
> ((1,2),3)
> [1,2,3]
```

44. Ask for the types of the expressions from Exercise 43. Note how the tuple types reflect each of the types of the components. Note also that the four final expressions all have different types.

45. Pairs are used quite often. Tuples with more components are less frequently. For pairs, there are projection functions. Try:

```
> fst (1,"Hello")
> snd (1,"Hello")
> fst (1,2,3)
> :t fst
> :t snd
> fst (snd (1,(2,3)))
```

Try to understand the type error and the types.

Currying

46. As indicated before, operators have types, too.

```
> :t (++)
```

In Exercise 8, I have explained that operators, if written between parentheses, can be used in prefix notation. In fact, if between parentheses, they're treated just like ordinary function names. Therefore, we can also ask for the type of an operator if using parentheses. The answer is

```
(++) :: [a] -> [a] -> [a]
```

Since `(++)` is a binary operator, it takes two arguments. Functions with multiple arguments are usually written in so-called *curried* style – after Haskell B. Curry, one of the first persons to use this style and also the person after whom the language Haskell was named. There is no magic here. Intuitively, it means that the function gets its arguments one by one, rather than all at the same time. It gets a list, then another list, and then produces yet another list as its result. This sequentiality is reflected in the (prefix) syntax of function application:

```
> (++) [1,2,3] [4,5]
> (++) "don't " "panic"
```

The parameters are just separated by spaces, there are no parentheses or commas to group the parameters together.

Concatenation is polymorphic again. It concatenates two lists of the *same* element type (the same variable is used everywhere), and the result list also has that element type.

47. Verify that concatenating two lists of different element type results in a type error. Try to understand the resulting type error message.

48. Here are some more functions with two arguments – all of them have types in the curried style. Check their types and try to find out what they are doing by passing them type-correct parameters. Also try to pass type-incorrect parameters and try to understand the error messages – for instance, try:

```
> :t take
> take 5 "Hello world!"
> take 42 "Hello world!"
> take 0 "Hello world!"
> take (-3) "Hello world!"
> take True "Hello world!"
> take 7 'H'
```

Perform similar tests for the following functions:

```
drop
replicate
const
```

What's the difference between `replicate 7 'x'` and `replicate 7 "x"`? Did you succeed passing type-incorrect parameters to `const`?

49. In the interpreter, it is possible to abbreviate expressions by giving them a name. The syntax for this is

`let` *identifier* `=` *expression*

The whole thing is called a *statement* and – provided that *expression* is type-correct, introduces *identifier* for further use. Try

```
> let hw = "Hello world!"
> :t hw
> hw
> length hw
```

Note that this way of binding identifiers in GHCi is slightly different from the way you define functions in separate Haskell modules.

50. Using `let` in GHCi, define an abbreviation of your own.

51. Recall Exercise 46, where we have introduced functions (or operators) with multiple arguments. I said that currying means that function parameters are passed one-by-one. This also means that not all parameters have to be provided at the same time. We can *partially apply* curried functions with multiple parameters:

```
> :t take
> :t take 2
> :t take 2 "Rambaldi"
> take 2 "Rambaldi"
> :t (++)
> :t (++) "Ramb"
> :t (++) "Ramb" "aldi"
> (++) "Ramb" " aldi"
> :t replicate
> :t replicate 3
> :t replicate 3 1
> :t replicate 3 'c'
> :t replicate 3 False
> replicate 3 False
```

52. We can abbreviate useful partial applications and give them a name, for instance:

```
> let dup = replicate 2
> :t dup
> dup 'X'
> dup 0
> dup True
> dup ' '
> let indent = (++) (dup ' ')
> indent "Hello world!"
> :t indent
```

53. Try the following expressions:

```
> take 3 "Hello"
> take False "Hello"
> take "x" "Hello"
> take 2.5 "Hello"
```

The first should succeed, the others fail. Try to explain why the others fail. Try to understand the error messages of the second and third case. The error message for the fourth expression is strange, and we'll explore this further.

Overloading

54 (medium, recommended)**.** Ask for the type of `2.5`:

```
> :t 2.5
2.5 :: (Fractional t) ⇒ t
```

This might come as a surprise. You might have expected to read `Float` here, for floating point number, or `Double`, for a double-precision number. Instead, the type contains a variable `t`, which suggests it's polymorphic, like many of the functions we've seen so far. But in addition, there's a so-called *constraint* in this type – the part before the double arrow ⇒. The way to read this type is as a logical implication: for all types `t`, if `t` is `Fractional`, then `2.5` has the type `t`. So, it's a bit like a polymorphic type, but with an additional condition on the type variable – we can't choose an arbitrary type, but instead must choose a `Fractional` type. Types that have such constraints are called *overloaded*.

55. What does `Fractional` mean? The identifier `Fractional` refers to a *type class*, that's a collection of types that share certain properties. You can ask the interpreter GHCi to give you more information about a type class and the types that belong to that type class:

```
> :i Fractional
```

The command `:i` (or `:info`) can be used with any identifier and results in information about where and how that identifier is defined. If used on a type class, you get to see the definition of the class, and information about which types belong to that class. In this case, you should see:

```
class (Num a) ⇒ Fractional a where
  (/) :: a → a → a
  recip :: a → a
  fromRational :: Rational → a
      -- Defined in GHC.Real
instance Fractional Double  -- Defined in GHC.Float
instance Fractional Float   -- Defined in GHC.Float
```

The first lines are the class declaration, which tells us something about the functionality that types of this class are required to support. But for now, we are more interested in the members of the class, which are called *instances*. In this case, the class consists of two types: `Double` and `Float`, double- and single-precision floating point numbers, respectively.

Numeric types and their classes

56. Recall

```
> :t 2.5
```

With what we've learned now, the type tells us that `2.5` can be a `Double` or a `Float`, depending on context, but not, for instance, an `Int`! Why not? Because `Int` is not an instance of the class `Fractional`! And this explains the strange type error message you get when trying

```
> take 2.5 "Hello"
```

The function `take` expects an `Int`, but is given something of type `(Fractional t) ⇒ t`, i.e., something that could be anything in the class `Fractional`. Now, `Int` is not in `Fractional`, hence the error.

57. Type classes are ubiquitous in Haskell, and there are many type classes predefined. If you check the type of numberic literals without a decimal dot, you'll get a different constraint:

```
> :t 2
```

The class `Num` is larger than `Fractional`, and comprises all numeric types, not just `Float` and `Double`, but also `Int` and `Integer`. Both are types for integers, but `Int` is bounded (but at least 32-bit large), while `Integer` can can compute with arbitrarily large integers (but is slightly less efficient). Type

```
> :i Num
```

to get information about the instances of `Num`.

58. Now, we can also look at the types of the numeric operators:

```
> :t (+)
> :t (*)
> :t (-)
> :t (/)
```

You see that all of them are overloaded. While the first three work for all numeric types, the division operator only works for fractional types.

59. The fact that numbers you type in, such as `2`, are overloaded over all numeric types (including the fractional types), means that you can use division on such numbers:

```
> 1 / 2
> 3 / 8
> :t 3 / 8
```

The result of such a division can still be used as a `Float` or a `Double`, but no longer as an `Int`. This also holds for divisions that happen to produce a whole number:

```
> 4 / 2
> :t 4 / 2
```

So,

```
> take (4 / 2) "Hello"
```

should produce a type error again. Verify that.

60. The types `Float` and `Double` are related (their only difference is their precision in positions after the decimal dot), so it makes sense that operations usually don't work on only `Float` or only `Double`, but instead are overloaded, with a constraint on `Fractional`.

Similarly, `Int` and `Integer` are related (their only difference is that `Int` is bounded and `Integer` isn't), and many operations work on both. So there's a class for these two types as well: `Integral`. Type

```
> :i Integral
```

and verify that `Integer` and `Int` are instances of `Integral`. The type class `Num` contains both the two integral types in `Integral` and the two fractional types in `Fractional`.

61. An operation on the integral types is integer division, called `div`:

```
> :t div
> div 4 2
> :t div 4 2
> div 7 2
> div 11 3
```

In integer division, the result is always rounded down to the nearest integer.

62. The counterpart to integer division is `mod`, which computes the remainder of integer division:

```
> :t mod
> mod 7 2
> :t mod 7 2
> mod 11 3
```

63. Recall that infix operators can be used as prefix functions by surrounding them with parentheses. Similarly, normal functions can be used as infix operators by surrounding them with backquotes. For some Haskell functions, such as `div` and `mod`, this is common practice:

```
> 11 `div` 3
> 11 `mod` 3
```

Printing values

64. Another type class is revealed by looking at the function `show`:

```
> :t show
(Show a) => a -> String
```

For all types `a` in `Show`, a value can be turned into a string. It turns out that many, many types are in `Show`. If you try

```
> :i show
```

you probably have to scroll up to be able to see all the instances in that class. Try it out with the types you already know, for instance:

```
> show 2
> show 2.5
> show True
> show 'a'
> show "Hello"
> show [1, 2, 3]
```

65. GHCi uses a variant of `show`, called `print`, internally when printing the results of expressions you type! If a type is not an instance of type `Show`, then GHCi doesn't know how to print a result.

Some types are not in `Show`, for example functions! If you try to evaluate a function, you get a type error:

```
> take
```

This results in:

```
No instance for (Show (Int -> [a0] -> [a0]))
  arising from use of `print'
Possible fix:
  add an instance declaration for (Show (Int -> [a0] -> [a0]))
In the expression: print it
In a 'do' expression: print it
```

The interesting part is the first line. There is no `Show` instance for the type `Int -> [a0] -> [a0]` – the type of `take`. In other words, GHCi doesn't know how to print a function on screen.

It is important to note that `take` itself is type correct, as

```
> :t take
```

demonstrates. The type error here only stems from the fact that GHCi tries to print the result of an expression you type in and implicitly computes `print it` (see Exercise 9) after evaluating the expression. That's why `print` is mentioned in the error message, even though you haven't typed it in.

Equality

66. Yet another interesting class is the class `Eq` of types supporting an equality operation:

```
> :t (==)
> :i Eq
```

The equality operator consists of two =-characters. Again, there are very many types supporting equality. The result is always a Boolean value:

```
> 1 == 1
> 1 == 1.5
> True == False
> 'x' == 'X'
> 'X' == 'X'
> [1, 2] == [2, 1]
> [1, 2] == [1, 2]
> "Hello" == "world"
```

67. Try to guess the result of

```
> False == (True == False)
```

and verify it in the interpreter. Can you explain what's happening?

68. Function types are an example of types not supporting equality. The expression

```
> (++) == (++)
```

does not return `True`, but results in a type error explaining that the type of the concatenation operator `[a] -> [a] -> [a]` is not an instance of the `Eq` class. The reasons for this are manifold: first of all, it is not quite clear when we should consider two functions equal? If they're mapping the same arguments to the same outputs? Or if they're using the same algorithm? The former is in general impossible to decide. The latter is often too restrictive, and would furthermore require that the implementation of a function is available for inspection at run-time, which is not the case in Haskell.

69. Find out the types of the following functions, and find out what they do by applying them to several arguments:

```
odd
even
gcd
sum
prod
```

70. The function `elem` checks whether a list contains a specific element. Check the type:

```
> :t elem
```

The function `elem` is often written infix as `` `elem` ``. Try to guess the answers before verifying them with the interpreter:

```
> 7 `elem` [2,3,5,7,11]
> 'e' `elem` "Hello"
> [] `elem` []
> [] `elem` [[]]
> [] `elem` [2,3,5,7,11]
```

Can you explain the type error?

Enumeration

71. Type

```
> [1..5]
```

and see what happens. Also try the following expressions

```
> [1,3..10]
> [10,9..1]
> ['a'..'d']
```

Recall that strings are just lists of characters.

72. Type

```
> [1..]
```

This will start printing all natural numbers starting with `1`. You will have to interrupt execution using `Ctrl+C`, i.e., by pressing the `Ctrl` key, holding it, then pressing `C`. You can do this whenever you want to interrupt GHCi, whether it is because the computation would not terminate or is just taking too long for your taste.

73. As exercise 71 shows, ranges can be specified for several types. The types that allow this are in yet another type class called `Enum`. Type

```
:i Enum
```

You will see that there are several methods in class `Enum`, among them `enumFromTo`, `enumFromThenTo` and `enumFrom`.

The range notation using `..` is so-called *syntactic sugar* for these functions. Haskell simplifies range expressions to applications of these functions. Verify that

```
enumFromTo 1 5
enumFromThenTo 1 3 10
enumFromThenTo 10 9 1
enumFrom 1
```

produce the same results as the range expressions above.

Defining new functions

74. Let's define a new function:

```
> let inc x = 1 + x
```

This is like defining an abbreviation, but additionally, we introduce a parameter `x` that we can use on the right hand side. This function increases a numeric value by 1:

```
> :t inc
```

The compiler infers the best possible type (including the `Num` constraint) for our function – you don't have to provide type information explicitly. The new function is the same as the function defined via partial application of `(+)`:

```
> let inc' = (+) 1
> :t inc'
> inc 41
> inc' it
```

75. Here is a function we couldn't have defined via partial application:

```
> let parens s = "(" ++ s ++ ")"
```

It surrounds a given string `s` with parentheses. Try it on a couple of strings. Also try it on the empty string.

76. See how

```
> init (tail "Hello")
```

drops the first an the last element of a string. Define a function `prune` that drops the first and the last element of any string. Check the type of your function:

```
> :t prune
```

It should be `[a] -> [a]`. Apply it to a list of numbers and see if it works as well. What happens if you apply it to a list of less than two elements? What happens if you apply it to a list of exactly two elements?

77. Here's a function `ralign` with two arguments:

```
> let ralign n s = replicate (n - length s) ' ' ++ s
```

Guess its type and what it does. Then verify in the interpreter. What does happen if `n` is smaller than the length of `s`?

Note that several parameters appear on the left hand side in a similar style as the function application would look: all arguments are separated by spaces, there are no parentheses or commas. The type of the function is in curried style (see Exercise 46), so it can be partially applied:

```
> let myralign = ralign 50
```

Test `myralign` on a couple of strings. What can you say about `length (ralign n s)` for arbitrary values of `n` and `s`?

78. Can you write a function `lalign` that moves a string to the left rather than to the right?

79 (medium)**.** Can you write a function `calign` that (approximately) centers a string rather than to move it to the left or right? Hint: You'll probably have to perform integer division via `mod` (see Exercise 62). Make sure that `length (calign n s)` is always `n`, and that the function works in all cases, whether `n` is even or odd, and whether `length s` is even and odd.

Anonymous functions

80. We have already seen that in Haskell, functions are just normal values. This goes even further: in Haskell, functions do not require a name! Just like you don't have to give a name to every number or string you use, you also can write down functions without giving them a name. Such functions are called *anonymous* functions. A named function such as

```
inc n = n + 1
```

can be written as

```
\n -> n + 1
```

The backslash \ and -> are just syntax: to introduce an anonymous function, and to separate the parameters from the body of the function. The above is the function that "maps `n` to `n + 1`". The backslash \ is read as "lambda" (for mathematical/historical reasons), so you can read the above function as "lambda `n` arrow `n + 1`". If you want to apply such an anonymous function, you have to put it in parentheses to delimit the function body: try

```
> (\n -> n + 1) 10
```

81 (medium)**.** For the range expressions introduced in exercise 71, we can use an anonymous function and the `:t` command to determine that ranges are indeed a feature that is tied to the type class `Enum`. Check

```
> :t \x y -> [x..y]
```

and try to understand the type.

82. Every function can be written as an anonymous function. For instance, if for some reason we wouldn't want to give `ralign` a name, we could use it anonymously:

```
> (\n s -> replicate (n - length s) ' ' ++ s) 10 "Hello"
```

83. On the other hand, of course, we can assign names to anonymous functions:

```
> let dec = \n -> n - 1
```

This means exactly the same as if we had written

```
> let dec n = n - 1
```

In fact, the latter is automatically translated into the former by the compiler. The latter syntax is provided as so-called *syntactic sugar*, i.e., a mechanism to make programming a little more convenient.

Higher-order functions

84. By now, you may (rightfully) ask what the big use of anonymous functions is at all, when all we might really want to do in the end is to assign a name to them again.

The answer lies – again – in the fact that functions in Haskell are more versatile than in many other languages. In particular, functions can be arguments to other functions. A highly useful example of such a function is `map`:

```
> map dec [1,2,3]
[0,1,2]
```

As you can see, `dec` (from Exercise 83) is applied to every element of the list. That's what `dec` does: it takes a function, and a list, and it applies the function to every element in that list. It is very similar to an iterator in many other languages.

A function such as `map` that takes other functions as arguments, is called a *higher-order function*.

85. For a function such as `map`, it turns out to be incredibly useful that we don't have to assign a name to every argument function. Try the following examples:

```
> map (\s -> " " ++ s) ["Hello","world"]
> map (\n -> n `mod` 2) [1,2,3,4,5,6,7]
> map (\n -> n * n) [1,2,3,4,5,6,7]
> map (take 2) ["Hello","world"]
> map not [True,False,True]
> map (\s -> (reverse (take 2 (reverse x)))) ["Hello","world"]
```

Think of more examples and try them out. Can you imagine what the type of `map` is? If you have a guess, try

```
> :t map
```

Did you guess right? If yes, you can be really proud of yourself. If not, no problem, just look at the examples again and try to understand the type.

86. Another higher-order function is `filter`. It takes a test, that is a function producing a Boolean value and applies it to all elements of a list. It only keeps the list elements for which the test evaluates to `True`. Here are a few examples (recall Exercise 69):

```
> filter even [1,2,3,4,5,6]
> filter (\x -> not (null x)) ["a","list","","of","strings",""]
> filter (\x -> x >= 3) [1,7,3,5,4,2]
> filter not [True,False,True]
> filter (\x -> x + 1) [1,2,3,4,5,6]
```

The operator `(>=)` just compares if a number is at least as large as the other. The last example will produce a type error. This situation may arise in a real program if you confuse `map` and `filter`, which are similar problems. Can you explain the type error? Look at the type of `filter` and at the type of the anonymous function for help:

```
> :t filter
> :t \x -> x + 1
```

87. What can you say about this function:

```
> let myfilter = filter (\x -> x True)
```

What is its type? How does it behave? Is it useful?

88. Look at types of the following higher-order functions. Apply arguments of the appropriate types to discover what these functions are doing:

```
dropWhile
takeWhile
all
any
```

89. Another higher-order function – in fact, an operator – is *function composition*, written as a dot in Haskell:

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

That's a complicated type, so let's rather look at the definition. In fact, we can define it at the interpreter prompt ourselves (thereby shadowing the already existing definition):

```
> let (.) f g = \x -> f (g x)
```

This operator takes two functions `f` and `g`, and it results in the function that, given an `x`, first applies `g` and then `f` to it. Try the following examples:

```
> (tail . tail) "Hello"
> map (head . tail . tail) ["Hello", "world"]
> ((\x -> x + 1) . (\x -> x * 3)) 42
```

Function composition can thus be used to sequence functions. The rightmost function in the sequence is applied first.

Operator sections

90. The final expression in Exercise 89 can actually be written much simpler. Try

```
> ((+1) . (*3)) 42
```

Similarly, try

```
> map (+1) [1..5]
```

An operator can be partially applied to its first or second argument if placed in parentheses. This construct is called an *operator section* and is a convenient form of syntactic sugar. In particular, `(+1)` is the same as writing `\x -> x + 1`.

Try also

```
> map (2^) [1..5]
> map (^2) [1..5]
> map (10-) [1..5]
> map (-10) [1..5]
```

and analyze the result. The final expression will cause a type error, because unfortunately, `(-10)` is not interpreted as an operator section, but as the negative number `10`. In this case, you have to write

```
> map (\x -> x - 10) [1..5]
```

or use the function `subtract`:

```
> map (subtract 10) [1..5]
```

91 (medium)**.** Try to predict the result of

```
> map ((-) 10) [1..5]
```

and verify your prediction.

92. Operator sections also work for backquoted function names. Try:

```
> map (`mod`3) [1..10]
```

93. Of course, operator sections are just normal expressions, so they have the same type as the anonymous functions they represent. Verify, for example, that

```
> :t (+1)
```

and

```
> :t \x -> x + 1
```

lead to the same result.

IO

94. In Exercise 65, I said that GHCi internally uses a function called `print` that is used to display results of evaluations on screen. It is similar to `show` in that it can produce output for anything that is in the `Show` class. But `show` just produces a `String`, whereas `print` actually displays it. Check the type of `print`:

```
> :t print
```

Turns out this is a function that takes any argument as long as it is in `Show`, as expected. But the result type is strange! It does not return a `String`, but an `IO ()`. You can read `IO ()` as an *IO action* yielding `()`. We'll learn more about these types now.

95. The type `()` is pronounced *unit*, and has a single value that is also written `()` and called *unit*, too. One way to think about this type is as a tuple with no elements. The main use of this type is as a parameter to other types, such as `IO` just above. IO actions are parameterized over their result type, but often, IO actions are not actually producing any interesting results (in particular if the only thing they do is produce output). We need a type to fill the parameter with, though, and better than to, say, use `Int` and agree that we always return `0`, is to use a dedicated dummy type such as `()`.

The type `()` is a normal type that can be used in other situations – try the following:

```
> ()
> ((),())
> [(),(),()]
> :t ()
> :t ((),())
> :t [(),(),()]
```

96. Values of type `IO ()` are IO actions that can be executed by GHCi. Whenever GHCi executes such an action, it ends up performing the IO contained within. We can make GHCi execute an IO action by typing in an expression that has an IO type:

```
> print True
> print 2
```

Note that GHCi acts in different ways depending on the type of the expression you provide it with: If the expression has IO type, it executes the action you have entered. If the expression has no IO type, it implicitly executes `print` on the expression you've entered. So as far as GHCi is concerned, you could have entered

```
> True
> 2
```

to produce the same effect as above. However, note the difference in the types:

```
> :t True
> :t 2
> :t print True
> :t print 2
```

97. Try

```
> print "Hello"
```

Note that this prints `"Hello"` on the screen, including the string quotes. This is because the idea of `print` is to produce a visualization of a value that is close to what you can

use in a Haskell program again. If you merely want to print text, then `putStrLn` is a better function – try

```
> :t putStrLn
> putStrLn "Hello world"
```

98. You can chain two IO actions using the operator `(≫)`. Try

```
> print True ≫ print False
> :t print True ≫ print False
```

99. Try

```
> let p = print True ≫ print False
```

Note that this doesn't print anything. The identifier `p` is bound to the action that prints `True`. Now try:

```
> :t p
> p
```

100. The function `sequence_` takes a list of IO actions and chains them all (note the underscore at the end). Try

```
> sequence_ [print 1, print 2, p]
```

101 (medium)**.** Of course, you're probably curious about the types of `(≫)` and `sequence_`, and have perhaps already tried

```
> :t (≫)
> :t sequence_
```

You'll see that `(≫)` is indeed a binary operator, and `sequence_` indeed takes a list as its argument. However, the type of these operators is actually overloaded, and any instance of class `Monad` works here. For now, the only thing you have to know is that `IO` is an instance of class monad. Replace the `m` in the types you see by `Monad` and the types should look more obvious.

102. The function `getLine` reads a line from the terminal. First try

```
> :t getLine
```

Note that `getLine` isn't actually a function in the sense that it takes no argument. It is an IO action, however, and one that does return a `String` rather than `()`, namely the string that has been read. Now try

```
> getLine
```

and note that you actually have to type in a line of text. After you finish, the line will be printed back to you by GHCi. This is because if the result of an IO action you type into GHCi is not of type `()`, GHCi again implicitly prints it.

103. We can chain IO actions even if they have different result types: try

```
> putStrLn "Please type a line:" ≫ getLine ≫ putStrLn "Thank you!"
```

Note that what you typed in is now lost: the result type of a chain of IO actions

```
> :t putStrLn "Please type a line:" ≫ getLine ≫ putStrLn "Thank you!"
```

is the type of the final action, here `()`.

Using `sequence_` is only possible if all actions have the same result type, because in a Haskell lists, all elements always have the same type. So verify that this fails

```
> sequence [putStrLn "Type a line:", getLine, putStrLn "Thank you"]
```

and that you understand the type error message.

104. Try

```
> "This is a line: " ++ getLine
```

This will fail with a type error. As we have already seen, IO actions have their own type, whereas `++` concatenates lists. But an `IO String` is no `String`. This clear distinction is it that enables us to manipulate IO actions with all our functional programming tools while having clear control about when the effects are actually executed. If we want to chain actions and use the result of one part in the next, we have to use the operator `(≫=)`, pronounced *bind*:

```
> getLine ≫= \x -> putStrLn ("Your line was: " ++ x)
```

Recall that once you execute this, you have to type a line. The string returned by `getLine` is passed to the right hand side of `(≫=)`. That is an anonymous function which picks up the string in `x` and uses it. Also verify the type of the overall action:

```
> :t getLine ≫= \x -> putStrLn ("Your line was: " ++ x)
```

105. Write an expression that reads a line and then prints it twice.

106. The bind operator allows us to extract the result of an IO action and use it in the rest of the computation. In GHCi, we can achieve something similar:

```
> x ← getLine
```

This will actually read a line, and bind the resulting string to `x`. Try:

```
> :t x
> x
```

Note how this is different from:

```
> let y = getLine
> :t y
> y
```

In essence, we decide whether we want to bind an identifier to the action or to the result of executing the action.

107. IO actions can be combined and can perform several inputs and outputs while executed. Can there also be IO actions that don't actually perform any input or output at all? Yes! The function `return` can achieve this. If you try

```
> :t return
```

you will once again see that it has an overloaded type. The main purpose of `return` is to combine previous intermediate results into a final result. Try:

```
> getLine >>= \x -> getLine >>= \y -> return (x ++ y)
> :t getLine >>= \x -> getLine >>= \y -> return (x ++ y)
```

The action reads two lines, and returns their concatenation.

108. Try

```
> let loop = putStrLn "Hello" >> loop
> :t loop
> loop
```

This will print `"Hello"` over and over, and you'll have to interrupt the execution to get back to the GHCi prompt.

109. Write a function that reads lines and prints them back repeatedly.

110 (medium)**.** Extend that function so that it maintains a counter that is incremented in every iteration and printed before asking for the next line.

Loading modules

111. So far, we have evaluated Haskell *expressions* in the interactive interpreter. We have also used `let` *statements* to assign names to functions. Furthermore, we have used special interpreter instructions (starting with a colon) to get more information.

We are now at a point where we move to full Haskell *programs*. Programs can contain expressions, but also so-called *top-level declarations*, for instance to declare modules, the visibility of functions, type classes or datatypes, and of course functions. Such top-level declarations cannot be entered interactively. On the other hand, interpreter commands such as `:t` are not part of the Haskell language and cannot be used in a program.

The simplest useful program consists of a single function. Open any text editor and create a text file called `Intro.hs`. Enter the following line:

```
inc x = x + 1
```

Then save and close the file. The file now contains the definition of a single function named `inc`, that adds `1` to its argument. Note that in a program, no `let` is required (and not even allowed) before the name of the function, but apart from that, the declaration of `inc` is equivalent to typing

```
> let inc x = x + 1
```

at the GHCi prompt.

As you will see, you have more possibilities to write programs within a file than interactively at the GHCi prompt. But the good thing is that you can still *test* your programs in GHCi. To make GHCi aware of your file, you have two possibilities: first, you can type

```
> :l Intro.hs
```

if the file `Intro.hs` is in your current working directory (otherwise you can give a full path name to that file). The command `:l` (for *load*) then loads the file into the interpreter.

If you start a new GHCi session, you can also pass the name of the file you're interested in as an argument to the GHCi command on the command line. Using the command line argument is more convenient if you are typically working with a command line. If you are starting GHCi by clicking on an icon, using the `:l` command is more convenient.

After loading the source file, verify that you can use the function `inc` defined therein by typing

```
> inc 41
```

Change the function `inc` in the file to increment by `2` rather than `1`. Then type

```
> :r
```

(short for `:reload`) to indicate that GHCi should reload the file, and try

```
> inc 41
```

again.

That's it for now. You are now familiar with a wide variety of Haskell language concepts, and well prepared to start defining your own functions.