Lecture Notes on Haskell Programming

Luca Padovani
Istituto di Scienze e Tecnologie dell'Informazione
Università di Urbino "Carlo Bo"

Generated on February 12, 2010

Comments, suggestions for improving the text, as well as typos and errors reports are very much appreciated. Please send them to `luca.padovani@uniurb.it`.

Contents

List of Tables

Part I

Foundations

Chapter 1

A taste of functional programming

Functional programming is an extremely powerful programming paradigm that can be fully mastered and appreciated only after years of practice. It is thus a difficult challenge for the author of an introductory text to convey enough excitement so as to encourage a student to depart from more familiar programming languages and explore with sufficient curiosity and devotion a new one.

The purpose of this chapter is to present a few examples of well-known algorithms implemented in a functional language, and to spot in them several of the features that will be subjects of more detailed chapters later on in the course. While the student is not expected to fully understand every single detail in the examples that follow, she will certainly appreciate the beauty, the conciseness, as well as the obvious correctness of these programs.

1.1 Functional Quick Sort

The introductory example we begin with is the well-known sorting algorithm that goes under the name of “quick sort”. The algorithm is a perfect application of *divide and conquer*: when the input to the algorithm is simple enough, the answer can be given immediately; when the input to the algorithm is complex, it can be partitioned into smaller sub-problems, each sub-problem is solved in isolation, and then all the solutions are combined to yield a solution to the original problem.

The idea behind the quick sort algorithm is simple: to sort (in ascending order) a bunch of data (we assume that the data is contained in a list) we proceed as follows: we spot one particular element in the list, which we dub the *pivot*. Next, we partition the elements of the list into two sub-lists: the sub-list of elements that are smaller than (or equal to) the *pivot*, and the sub-list of elements that are greater than the *pivot*. Not counting the occurrence of the *pivot* element, both the two sub-lists thus determined are *smaller than* (in fact, shorter) that the original list we started from. We can then sort (using the very same quick sort algorithm) both sub-lists, and concatenate them in the correct order (together with the *pivot* element) in order to produce the sorted list.

In Haskell, quick sort can be implemented as follows:

```
qsort :: (Ord a) ⇒ [a] → [a]
qsort [] = []
qsort (x : xs) = qsort l1 ++ [x] ++ qsort l2
    where
    l1 = filter (<= x) xs
    l2 = filter (> x) xs
```

Before explaining some of the details, let us re-read the algorithm in light of the code just presented. This code snippet defines a function called `qsort`. The function is given a *type* in the first line. Here, the type declares that the function accepts a list of elements of type `a` (the type `[a]` before the → symbol) and returns a list of elements of type `a` (the type `[a]` after the → symbol). The → symbol provides the intuition that something (of type `[a]`) enters the function and something else (again of type `[a]`) is produced by the function as a result. The context (**`Ord`** `a`) roughly means that the elements of type `a` must admit a reasonable notion of ordering. Given that we are implementing a sorting algorithm (over elements of type `a`) this is, after all, a reasonable requirement.

The second line of the code snippet simply states that when sorting the empty list (the `[]` symbol on the left hand side of = is the input to the function), the result is the empty list (the `[]` symbol on the right hand side of the = is the output of the function). Thus, the second line provides a solution to the sorting problem, when the input of the function is simple enough.

The rest of the code snippet is where the interesting things happen. The third line defines the behavior of the `qsort` function when the input is a nonempty list. This is stated in the `(x : xs)` pattern on the left hand side of =. The pattern is to be read as "the input is a nonempty list whose first element is `x` and whose remainder list is `xs`". In this case we choose `x` as the *pivot* element and we produce the result `qsort l1 ++ [x] ++ qsort l2`. Namely, we sort `l1` and `l2` and we concatenate (by means of `++`), the (sorted) `l1` followed by the one-element list `[x]` containing the *pivot* element followed by the (sorted) `l2`. The result is correct provided that `l1` is the sublist containing only those elements of `xs` that are smaller than (or equal to) `x` and that `l2` is the sublist containing only those elements of `xs` that are greater than `x`. This is indeed the case, as `l1` and `l2` are defined right after the **`where`** keyword as **`filter`** `(<= x) xs` and **`filter`** `(> x) xs`. These two expressions just do what they intuitively mean when read as plain text: **`filter`** `(<= x) xs` is the sublist (**`filter`**) containing only those elements of `xs` that are smaller than (or equal to) the *pivot* `x` (`(<= x)`). Symmetrically for `l2`.

It could be argued that much of the conciseness of this implementation of "quick sort" derives from the use of the two **`filter`** expressions, which conceals a great deal of code. However, **`filter`** is *not* a Haskell primitive instruction. It is itself a plain function that can be naturally defined in a similar fashion, as follows:

```
filter :: (a → Bool) → [a] → [a]
filter _ [] = []
filter p (x : xs) | p x = x : filter p xs
                  | otherwise = filter p xs
```

Roughly, the **`filter`** function is a two-arguments function where the first argument is a "predicate" and the second argument is a list of elements. The function returns the list of all and only elements of the original list that satisfy the predicate. The type given to **`filter`**, albeit puzzling, tells us that the first argument of **`filter`** has type `(a → Bool)`, namely it is itself a function that takes an argument of type `a` and returns a value of type **`Bool`** (a boolean value) indicating whether the element satisfies or not the predicate. The second argument of **`filter`**, as expected, has type `[a]`, namely it is a list of elements of type `a`.

How do we filter elements? Once again there is a case that is easy enough to be solved immediately. This is when the input list is empty, in which case we can only return the empty list as result (second line). Observe that in this case the predicate plays no role, hence the symbol `_` on the left hand side of `=` to indicate that this argument is unused in this equation. When the list is not empty (third line), namely when it has a first element `x` followed by a list `xs`, we distinguish two further sub-cases. If `x` satisfies the predicate, namely if `p x` is "true", then `x` is preserved in the result and is followed by all the elements in `xs` that satisfy the same predicate. If `x` does not satisfy the predicate, we simply return all the elements in `xs` that satisfy the same predicate.

The **`filter`** function is so useful that it is included in the Haskell standard library together with dozens of other general-purpose functions and data types.

We conclude this section with a summary of the features that Haskell provides and that have been implicitly or explicitly used in the coding examples above.

automatic memory management Both `qsort` and **`filter`** do manipulate and create lists of arbitrary length and yet there is no explicit evidence of "instructions" that allocate and free memory used by the program. All the allocations and deallocations are taken care of automatically by the Haskell runtime. Memory is allocated when necessary, and is deallocated when the Haskell runtime detects that it is no longer useful.

lack of pointers The use of *patterns* saves us from using memory addresses for referring to the elements of the list.

pattern matching Data structures such as lists are examined by comparing them against *patterns* (on the left hand side of `=`) in both the functions above. We will see pattern matching at work thoroughout the whole course, and will focus specifically on it in Chapter 11.

type inference For both functions above we have explicitly provided their type as a declaration. However, this is not necessary as Haskell is able to figure out automatically the type of (almost) every expression. This is done through a process called *type inference* (although the term *type reconstruction* might be more appropriate). Observe that we have *not* provided explicit type annotations for all the other identifiers (such as `x`, `xs`, `l1`, and `l2`). Haskell has inferred their type automatically.

expressive type system The context annotation (**`Ord`** `a`) suggests that Haskell's type system accounts for more expressive types than those found in more common programming languages.

declarative style Haskell encourages a declarative (as opposed to "sequential") way of defining functions. This is testified by the very use of pattern matching, but also by the use of *guards* (in the definition of **`filter`**, for distinguishing the subcases of the case when the input list is not empty) and by the syntax of the **`where`** clause, which postpones the definition of names as in mathematical writings.

polymorphism Both `qsort` and **`filter`** are *polymorphic* functions dealing with a *polymorphic* data type (the list). Polymorphism (meaning "many forms") identifies the ability of dealing with different data types in one single definition. In fact, **`filter`** above works with lists whose elements have an arbitrary type (the type variable `a` roughly means "any type"). Similarly, `qsort` works with lists whose elements have arbitrary type `a`, provided that the type is equipped with a notion of "ordering" (see the context (**`Ord`** `a`)).

overloading The operators <= and > used in the definition of qsort can compare elements of arbitrary type a. However, their exact definition depends on the specific type a. Haskell is able to pick just the right definition of these operator according to the context. Namely, these are *overloaded* operators.

higher-order functions The **filter** function accepts a predicate as its first argument. But predicates, as the type of **filter** reveals, are nothing but functions of type (a → **Bool**) (in this case). Namely, **filter** is a function that accepts another function as argument. The ability to treat functions as first-class citizens is in fact the single feature that gives the name to the functional programming paradigm. It provides the programmer with a tremendous abstraction tool that helps writing reusable software.

strong typing Haskell is a *strongly typed* language. This means that a program is evaluated only if it passes a large number of tests that verify its correctness to a certain extent. It is often said that “well-typed programs in a strongly typed language cannot go wrong”. While this is not true in practical terms, the well-typedness of a functional program provides much stronger guarantees about the program than in the other programming languages. For example, it is impossible for a well-typed Haskell program (not using “unsafe” language features) to abruptly terminate because of some invalid memory access or because of the misuse of a data structure.

1.2 The interactive evaluation environment

In order to try the quick sort algorithm defined earlier we have two ways: either we compile it to an executable program by means of a *Haskell compiler*, or we load it into an *interactive Haskell interpreter*. Here and in the rest of the course we will use the GHCi interactive interpreter,[1] which is started with the command ghci followed by the name of the Haskell script containing the qsort function, in this case qsort.hs:

```
$ ghci qsort.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( qsort.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

The Main> prompt informs us that the environment is waiting for expressions to be evaluated in the context of the Main module. When no explicit module name is given, as in our qsort.hs script file, Haskell assumes a default Main. We will talk more about modules in Chapter 17.

We can now test the qsort function by applying it to a list of numbers

```
> qsort [9,8,7,6,5,4,3,2,1,0]
[0,1,2,3,4,5,6,7,8,9]
```

and GHCi answers with the resulting list, which is correctly sorted.

As we said, qsort is polymorphic and can be applied to lists of elements whose type is arbitrary. Since Haskell represents strings as lists of characters, we can ask qsort to sort the characters in a list according to their Unicode order:

[1]Available at http://www.haskell.org/ghc/. Another interpreter called HUGS, available at http://www.haskell.org/hugs/, can be used as well.

```
> qsort "hello,␣world!"
"␣!,dehllloorw"
```

Observe that the `qsort.hs` script we have loaded only contains the definition for the `qsort` function, and that the **`filter`** function is used but not defined. Like we said **`filter`** is defined in the standard Haskell library and we can use it right away:

```
> filter (<= 'h') "hello,␣world!"
"he,␣d!"
> filter (> 'h') "hello,␣world!"
"lloworl"
```

Suppose now we want to create a standalone Haskell program that reads a string of characters, sorts the characters according to their Unicode order, and prints the result on the terminal. We need to tell Haskell what is the entry point of our program, and what to do when the program is started. This is accomplished by adding a definition for the name `main` to the Haskell script that defined `qsort` above:

```
main :: IO ()
main = do s ← getContents
          putStrLn $ qsort s
```

The declaration `main :: IO ()` states that `main` is an input/output action (**`IO`**) that does not produce anything (`()`). Then, the string is read by means of action **`getContents`**. In the sequel, the string is sorted with `qsort` and printed on the terminal followed by a carriage-return character by means of **`putStrLn`**.

To compile this Haskell script into an executable file, just run the `ghc` Haskell compiler from the command-line prompt:

```
$ ghc qsort.hs
```

Absence of messages denotes the fact that compilation was successful. At this point an executable file called `a.out` has appeared in the same directory from which we launched `ghc`. When running `a.out` we provide some text and inform that the text is over by hitting `^D` on the keyboard[2]:

```
$ ./a.out
hello, world!
^D
 !,dehllloorw
```

[2] `^Z` under Windows.

Chapter 2

Basic data types

Every program manipulates data. In this chapter we get acquainted with the most frequently used primitive data types that Haskell provides. We will also clarify the distinction between expressions and values and informally introduce the concept of *reduction* or *evaluation*.

2.1 Expressions and values

We start by using the interactive environment as a simple calculator. At the prompt we can enter an *expression* and ask the environment to *evaluate* it:

```
> 1 + 2 * 3
7
```

Here we have entered the expression `1 + 2 * 3` that denotes the sum of the constant number `1` with the product of the two numbers `2` and `3`. As is common in most programming languages, `*` denotes multiplication. The environment has evaluated this expression to `7`, as expected. More precisely, the environment has *reduced* the expression through a number of subsequent *reduction steps*, indicated by $\Longrightarrow$ in the following representation:

`1 + 2 * 3` $\Longrightarrow$ `1 + 6` $\Longrightarrow$ `7`

The number `7` is a particular expression that cannot be reduced further, and is thus given the name of *value*.

Observe that evaluation first reduces the subexpression `2 * 3` to the value `6`, and then reduces the expression `1 + 6` to the value `7`. This means that the `*` operator has precedence over the `+` operator, as expected.

2.2 Integer numbers

Integer numbers can be entered using the standard syntax, where negative numbers are preceded by the `-` symbol:

```
> 3
3
> -5
-5
```

Operator	Description
m + n	addition
m - n	subtraction
-m	negation
m * n	multiplication
m ^ n	exponentiation m^n

Function	Description
abs m	absolute value
div m n	integer division
mod m n	integer modulus
negate m	negation

Table 2.1: Common operators and functions over numbers.

Haskell also understands numbers written in base 16 (when they are preceded by `0x`) and base 8 (when they are preceded by `0o`):

```
> 0x1F
31
> 0x10
16
```

Table 2.1 lists some predefined operators and functions over numbers. All operators except for negation are used in infix form, pretty much as in most other programming languages. Thus we will write:

```
> 2 + 3
5
> 2 - 3
-1
> 2 * 3
6
> 2 ^ 3
8
```

Functions are *applied* by listing their arguments next to their names, separated by spaces:

```
> negate 5
-5
> div 5 2
2
> mod 5 2
1
```

Compare this with the standard mathematical notation. In particular, in mathematics one usually writes $f(a) + f(b)$ whereas the same expression in Haskell would be

```
f a + f b
```

since function application has precedence over (is stronger than) the operator +. For this reason, parentheses are necessary in case we want to express $f(a + b)$, which would be written thus:

```
f (a + b)
```

As a matter of facts, in this last example the space separating f from its argument is not stricly necessary, as the parenthesis makes it clear where the function name ends and where the argument starts. However, we will always separate arguments with spaces for uniformity.

Observe that the use of no explicit symbol for function application is a first evidence that functions and function application are so frequent in a functional program that they deserve the lightest syntax, and there is no syntax that is lighter than no syntax at all!

It is possible to apply an operator as if it were a function, by writing the operator within parentheses:

```
> (+) 2 3
5
> (*) 2 3
6
```

It is also possible to use a (two-argument) function as if it were a (binary) operator, by writing the function name within backquotes:

```
> 5 ‘div‘ 2
2
> 5 ‘mod‘ 2
1
```

Beware when composing larger expressions using unary operators, as subtle problems may arise:

```
> abs -5
<interactive>:1:0:
    No instance for (Num (a → a))
      arising from a use of ‘-’ at <interactive>:1:0-5
    Possible fix: add an instance declaration for (Num (a → a))
    In the expression: abs - 5
    In the definition of ‘it’: it = abs - 5
```

Here the intention is to compute the absolute value of `-5`, but `GHCi` interpreted the expression as **`abs`** `- 5`, namely as a subtraction of `5` from **`abs`** which makes no sense.

In this case the error message is quite obscure, and the suggested fix is also misleading! It turns out that in Haskell (and in most functional languages), many apparently syntactic errors show up as *typing errors*. In other words, the expression **`abs`** `-5` is syntactically well-formed, but it cannot be given a meaning. We will discuss more about types and meaning in Chapter 3. For now, let us note that the correct way to write the expression above is to use parentheses around `-5`, thus:

```
> abs (-5)
5
```

In general, parentheses can be used to explicitly group subexpressions and to force a particular evaluation order. Thus `(1 + 2) * 3` is evaluated as follows:

```
(1 + 2) * 3 ⟹ 3 * 3 ⟹ 9
```

2.3 Fractional numbers

Numbers with a fractional part can be entered with the usual notation:

```
> 1.0
1.0
> 1.5
1.5
```

but observe that Haskell requires both the integral and the fractional part to be specified:

```
> .5
<interactive>:1:0: parse error on input '.'
> 2.
<interactive>:1:2: parse error (possibly incorrect indentation)
```

The exponential notation is also available

```
> 1.5e-7
1.5e-7
> 1.5e7
1.5e7
```

where `1.5e-7` denotes the number 1.5×10^{-7} and `1.5e7` (or equivalently `1.5e+7`) denotes the number 1.5×10^{7}. All of the operators shown in Table 2.1 are available with the expected semantics:

```
> 1.5 + 2 * 3.5
8.5
> (1.5 + 2) * 3.5
12.25
> 0.5 ^ 10
9.765625e-4
```

Moreover, fractional numbers have a division operator / that, unlike integer division **`div`**, yields possibly fractional results. Compare

```
> 7 / 2
3.5
```

and

```
> 7 `div` 2
3
```

Table 2.2 lists some of the predefined operators and functions for fractional and floating-point numbers.[1]

```
> 1.0 / 0.0
Infinity
> -1.0 / 0.0
-Infinity
> 0.0 / 0.0
NaN
```

2.4 Booleans

Haskell provides two constants representing *truth values*:

[1]We will see that Haskell makes a distinction between fractional numbers and floating-point numbers. For the time being we will use the terms "fractional" and "floating-point" as synonims and postpone a more precise treatment to the next chapter.

Constant	**Description**
`pi`	approximation of π

Operator	**Description**
m `/` n	division
m `**` n	exponentiation m^n

Function	**Description**
`recip` m	reciprocal
`exp` m	exponential e^m
`sqrt` m	square root $\sqrt{m}$
`log` m	natural logarithm $\ln m$
`logBase` m n	logarithm $\log_m n$
`sin` m	sine
`cos` m	cosine
`tan` m	tangent
`asin` m	inverse sine
`acos` m	inverse cosine
`atan` m	inverse tangent
`sinh` m	hyperbolic sine
`cosh` m	hyperbolic cosine
`tanh` m	hyperbolic tangent
`asinh` m	inverse hyperbolic sine
`acosh` m	inverse hyperbolic cosine
`atanh` m	inverse hyperbolic tangent

Table 2.2: Common operators and functions over fractional and floating-point numbers.

Operator	Description
m == n	equal to
m /= n	not equal to
m < n	less than
m > n	greater than
m <= n	less than or equal to
m >= n	greater than or equal to

Table 2.3: Relational operators.

```
> True
True
> False
False
```

The *conjunction* of two boolean values, denoted by the binary operator &&, is **True** only if both values are **True**:

```
> True && True
True
> True && False
False
```

The *disjunction* of two boolean values, denoted by the binary operator ||, is **True** if either one of the values, or both, are **True**:

```
> True || False
True
> False || False
False
```

Logical negation is denoted by prefix operator **not**:

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

Boolean values may result from comparing numbers with each other, to determine whether a number is equal to/different from/smaller than/greater than another one. Table 2.3 lists the Haskell relational operators.

```
> 2 < 3
True
> 2 == 3
False
```

The relational operators are *overloaded*, meaning that they are able to seamlessly compare different data types. For example we have:

```
> 2.0 < 3.0
True
> 2.0 == 3.0
False
> False < True
True
```

```
> True < False
False
```

Note however that both operands of a relational operator must have the same type:

```
> 1 == True
<interactive>:1:0:
    No instance for (Num Bool)
      arising from the literal ‘1’ at <interactive>:1:0
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of ‘(==)’, namely ‘1’
    In the expression: 1 == True
    In the definition of ‘it’: it = 1 == True
```

Once again the error message appears not particularly informative, but only because we still lack some understanding of the mechanisms behind types and overloading. We will come back to these topics in the next chapters.

Boolean values are most often used to “take decisions”. For example, it is often the case that the value of an expression depends on some condition. Such an expression is called a *conditional expression* and is denoted in Haskell by the three keywords **if**, **then**, and **else**:

if B **then** E_1 **else** E_2

Here B is a (boolean) expression that determines the condition to be verified. If B evaluates to **True**, then the whole expression reduces to E_1. If B evaluates to **False**, then the whole expression reduces to E_2. For example:

```
> if 2 < 3 then 1 else -1
1
> if 2 > 3 then 1 else -1
-1
```

Unlike many other programming languages, a conditional expression is itself an expression, thus it can be used for building larger expressions:

```
> (if 2 < 3 then 1 else -1) * 2
2
> (if 2 > 3 then 1 else -1) * 2
-2
```

The branch following **else** in a conditional expression whose condition evaluates to **True** is not evaluated. Symmetrically, the branch following **then** in a conditional expression whose condition evaluates to **False** is not evaluated. Thus, the expression

```
> if True then 1 else 1 ‘div‘ 0
1
```

does not raise an error despite the **else** branch attempts a division by zero.

Logical conjunction and disjunction operators can be expressed in terms of conditional expressions as follows:

E_1 && E_2 = **if** E_1 **then** E_2 **else** **False**
E_1 || E_2 = **if** E_1 **then** **True** **else** E_2

In particular, both && and || are *short-circuited* and evaluate their arguments only if necessary, from left to right. Thus we have:

```
> False && 1 / 0 == 1
False
```

```
> True || 1 / 0 == 1
True
> True && 1 / 0 == 1
False
> True && 1 `div` 0 == 1
*** Exception: divide by zero
> False || 1 `div` 0 == 1
*** Exception: divide by zero
```

2.5 Exercises

Exercise 2.1. Evaluate the following Haskell expressions “by hand”. Use the interactive Haskell environment to check that your answers are correct.

1. `1 / 2 + 3`
2. `(1 / 2) + 3`
3. `1 / (2 + 3)`
4. `sin pi ** 2`
5. `False || False && True`
6. `not False && False`
7. `not (if True then False else True)`
8. `if if False then True else False then False else True`
9. `(if pi < pi / 2 then 1 else 2) + 1`
10. `pi ** 2 < 3 || pi ** 2 >= 3`

Exercise 2.2. Write Haskell expressions corresponding to the following mathematical formulas and evaluate them in `GHCi`:

1. $\sqrt{2^2 + 3^2}$
2. $\sin^2 \frac{\pi}{2}$
3. $|-100 + 2^{2+3}|$
4. $\sqrt{2\pi 10}\left(\frac{10}{e}\right)^{10}$
5. $\dfrac{\frac{\pi}{2} + \frac{3}{4}}{\sqrt{2}}$

Exercise 2.3. Try to make understand the error messages resulting from the evaluation of the following Haskell expressions:

1. `True == True == True`
2. ``4 `div` -2``

3. `abs True`
4. `if 3 then 1 else 2`
5. `if True then 1 else False`
6. `3 < True`

Exercise 2.4. Add parentheses so that the following expressions do evaluate correctly:

1. `sin sin 3`
2. `True < 2 < 3`
3. `div 8 div 4 2`
4. `(+) negate 5 2`
5. `mod (+) 1 (*) 3 4 div 16 4`

Chapter 3

Types and classes

Haskell is a *strongly typed programming language*, meaning that every Haskell script is checked against a large number of consistency rules before it can be executed. If a script successfully passes all the checks, then it is guaranteed to be free from lots of errors, such as the application of functions to the wrong number of arguments and the inconsistent use of operators against unexpected operands. An informal knowledge about types, classes, and the consistency rules is fundamental for understanding the semantics of Haskell programs and for efficiently decoding, spotting, and fixing error messages signalled by the Haskell compiler.

3.1 Types as sets of values

A *type* represents a collection of related values. For example, the set of boolean values `False` and `True` is represented by the type `Bool`. Among the predefined data types, Haskell also provides the programmer with the types `Int` and `Float`, which represent integer and (single-precision) floating-point numbers, respectively.[1]

Intuitively we can think of types as of sets of values:

```
Bool  = { True, False }
Int   = { ..., -5, -4, ..., -1, 0, 1, 2, ..., 4, 5, ... }
Float = { ..., -4.5, ..., -0.5, ..., 0.0, ..., 1.0, ... }
```

Observe that `Bool`, `Int`, and `Float` are *disjoint types* in the sense that they share no common value. In other words, there is no value that is both in the set `Int` and also in the set `Bool`, and symmetrically there is no value in the set `Bool` that is also in the set `Int`. A similar relation also holds for `Int` and `Float` and for `Float` and `Bool`. For this reason, when we say "the value v has type `T`" we intuitively mean that v belongs to the set `T` and no ambiguity may arise because of the disjunction property of Haskell types.

We can extend the notion of "type" from values to expressions. We say that an expression E has type `T` if, whenever E reduces to a value v, we have that v has type `T`. For example, the value `True` has type `Bool` and so does the expression `True || False`.

We can ask `GHCi` about the type of an expression by using the `:type` command as follows:[2]

```
> :type True
```

[1]Haskell also provides a double-precision floating-point type called `Double`.

[2]The `:type` command can also be abbreviated as `:t`.

```
True :: Bool
> :type True && False
True && False :: Bool
> :type 1 == 2
1 == 2 :: Bool
```

The :: symbol is normally read “has type”.

3.2 Arrow types

In Haskell functions do have a type. This fact has far more consequences than those that can be grasped at this early stage. We can ask GHCi the type of the negation function for boolean values:

```
> :type not
not :: Bool → Bool
```

The fact that **not** is a function is evident from its type, which is a so-called *arrow type*. A type of the form $S \to T$ denotes a function whose domain is S and whose codomain is T. Namely, the function accepts arguments of type S and produces results of type T. In the example above, both the domain and the codomain of **not** are **Bool**. Namely, **not** accepts boolean values and produces boolean values.

When asking GHCi about the type of a multi-argument function, say logical conjunction (&&), we obtain the following:

```
> :type (&&)
(&&) :: Bool → Bool → Bool
```

As a rule of thumb, the number of “arrows” is the number of arguments accepted by the function. In an arrow type $S_1 \to S_2 \to \cdots \to S_n \to T$ the type S_i is the type of the i-th argument, while T is the type of the value returned by the function. It will take a while to fully appreciate what this type really means, and all the implications it has.

3.3 Type classes

A *class* represents a collection of related types. Roughly speaking, the relation between types in the same class is given by a set of operations that are available for values of these types. For instance, the type class **Eq** represents all Haskell types for which a notion of “equality” exists. All the basic types mentioned above – **Bool**, **Int**, and **Float** – are *instances* of the class **Eq**, namely all the values of type **Bool**, **Int**, and **Float** can be compared for (in)equality using the operators == and /=. On the other hand, arrow types are *not* instances of the class **Eq**, namely it is not possible to ask whether two functions are equal or not.[3]

The fact that a type is an instance of a class is denoted by writing the class followed by the type, as follows:

Eq Bool

If we ask GHCi about the type of the equality operators, we obtain the following:

```
> :type (==)
(==) :: (Eq a) ⇒ a → a → Bool
```

[3]Mathematically speaking it does make sense to ask whether two functions are the same, but this kind of equality between functions in undecidable and thus not interesting in the context of programming.

The actual type of `(==)` is `a → a → Bool`, which follows the ⇒ symbol. In this type, `a` is a so-called *type variable* which can be roughly thought of as "any type". The context `(Eq a)`, which precedes the `=>` symbol, provides class information for the type variables that occur in the actual type. In this case, `(Eq a)` informs us that, in order to be able to apply the `==` operator to values of type `a`, the type `a` must be an instance of the class `Eq`. That is, `==` can be used for comparing values of arbitrary type, provided that the type is an instance of `Eq`. Since `Bool`, `Int`, and `Float` are all instances of `Eq`, we can use `==` for comparing boolean, integer, and floating-point values:

```
> True == True
True
> 1 == 1
True
> 1.0 == 1.0
True
```

However, we cannot use `==` for comparing functions:

```
> not == not
<interactive>:1:0:
    No instance for (Eq (Bool → Bool))
      arising from a use of ‘==’ at <interactive>:1:0-9
    Possible fix: add an instance declaration for (Eq (Bool → Bool))
    In the expression: not == not
    In the definition of ‘it’: it = not == not
```

`GHCi` informs us that `Bool → Bool`, which is the type of `not`, is not an instance of the class `Eq`.

3.4 Number literals

We have stated earlier that `Int` and `Float` are *disjoint types*. It may be argued, however, that this is not the case. For example, while it is true that the value `1` and the value `1.0` are syntactically different, it would seem natural, mathematically speaking, to consider them as *the same number 1*. To solve the puzzle, we must be careful distinguishing values from the literals we use to denote them. In practice, integer and floating-point values do have entirely different representations when stored in the main memory or in a register of the CPU. As a consequence, the integer value `1` and the floating-point value `1.0` *are* different and have type `Int` and `Float` respectively. However, when we write the literal `1` in a Haskell expression, which value do we mean? Most programming languages follow a rather pragmantic approach: if the literal contains no `.` symbol, as is the case of `1`, then the literal denotes an integer value; if the literal contains a `.` symbol, as is the case of `1.0`, then it denotes a floating-point value.

The Haskell type system is expressive enough to permit a more flexible treatment of literals. When asking `GHCi` about the type of `1`, we obtain the following:

```
> :type 1
1 :: (Num t) ⇒ t
```

By looking at the type of `1` we realize that Haskell provides another type class called `Num` whose purpose is to collect all those types representing numbers. Both `Int` and `Float` are instances of `Num`, but the type `Bool` is not. In other words, `GHCi` infers that `1` is a "number", although it does not know precisely which type – either `Int` or `Float` – of number. The precise type is determined by looking at the context where the literal 1 occurs: if the literal occurs in a context where an integer value is expected, then the literal 1 is understood to denote the integer value 1; if the literal occurs in a context where a floating-point value is expected, then the literal 1 is understood as the floating-point value 1.0. When the literal occurs alone, as in the example above, there is no context that can help GHCi figuring out whether 1 stands for an integer or for a floating-point value, hence the resulting type (Num t) ⇒ t.

The Num type class is characterized by the +, -, and * operators, along with a few more functions such as negate. Indeed we have

```
> :type negate
negate :: (Num a) ⇒ a → a
```

and

```
> :type (+)
(+) :: (Num a) ⇒ a → a → a
```

Namely, negate can be applied to every value whose type is an instance of Num, and + can be applied to every pair of values whose type is an instance of Num.

3.5 Subclasses

Classes can be organized as a hierarchy: we may define a type class S that is a *subclass* of a type class T. In this way, all the types that are instances of S are also instances of T and all the methods that are available in the class T are available also in all of its subclasses, including S.

For example, in the Haskell library the type class Num has two subclasses: Integral is the subclass of those numeric types without any fractional part, whereas Fractional is the subclass of the non-integral, numeric types. In turn, the type class Fractional has a subclass Floating for floating-point numbers. The type Int is an *instance* of the class Integral and thus, indirectly, of the class Num. The type Float is an instance of the class Floating and thus, indirectly, of the class Fractional and ultimately of the class Num.

At last we can fully appreciate the difference between the literal 1 and the literal 1.0:

```
> :type 1
1 :: (Num t) ⇒ t
> :type 1.0
1.0 :: (Fractional t) ⇒ t
```

The literal 1 contains no . symbol, so it may well stand for both an integral or a fractional value and this is reflected in its type, where the type variable t belongs to the type class Num which is the superclass of all numeric types. The literal 1.0 does contain a . symbol, so it surely belongs to a non-integral type. Once again the type provides us with such information.

This brief introduction to Haskell's class hierarchy may also help explaining the reason why Haskell apparently provides different operators and functions that appear to do basically the same job. The type of the binary function div is

```
> :type div
div :: (Integral a) ⇒ a → a → a
```

that is **div** is the division between values of any type a that is instance of the **Integral** class. Since such values do not account for a fractional part, this division necessarily truncates the result.

On the other hand, the (/) function has type

```
> :type (/)
(/) :: (Fractional a) ⇒ a → a → a
```

namely / is the division between values of any type a that is instance of the **Fractional** part.

Finally, observe that the trigonometric functions are applicable only to numbers in the **Floating** class:

```
> :type sin
sin :: (Floating a) ⇒ a → a
```

3.6 Type consistency

Not every expression is evaluated by GHCi, only the *well-typed* ones are. The precise rules for deciding whether an expression is or isn't well typed are fairly complex and involve lots of technical issues that go well beyond the scope of this introductory course. However, a basic understanding of the well-typedness rules is essential for interpreting GHCi error messages and for fixing ill-typed expressions.

The fundamental idea is that, when we apply a function to an argument, GHCi checks that the type of the argument we supply and the type of the argument expected by the function do coincide. If this is the case, GHCi declares that the whole expression is well typed, and its type is the type returned by the function. For example, **not** expects a **Bool** and returns a **Bool**. Furthermore **True** has type **Bool**. Namely, **not True** is a well-typed expression that has type **Bool**. If we try to apply **not** to some value that is not of type **Bool**, we obtain a *type error*, although probably not the one we expect:

```
> not 1
<interactive>:1:4:
    No instance for (Num Bool)
      arising from the literal ‘1’ at <interactive>:1:4
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of ‘not’, namely ‘1’
    In the expression: not 1
    In the definition of ‘it’: it = not 1
```

The error message originates from two facts: (1) GHCi expects the argument of **not** to have type **Bool** and (2) the type of 1 is (**Num** t) ⇒ t. In order for these two types to be the same type, it must be t = **Bool**. However, it appears that **Bool** is not an instance of class **Num** (indeed, booleans are not numbers), and this generates the error message above.

Conversely, no problem arises if we use the operator + with numbers:

```
> :type (+) 5 2
(+) 5 2 :: (Num t) ⇒ t
```

Recall that (+) has type (**Num** a) ⇒ a → a → a and that both 5 and 2 have type (**Num** t) ⇒ t. Then, (+) 5 2 must have type (**Num** t) ⇒ t.

Since + works with generic "numbers", it works with arguments whose type belongs to a subclass of **Num**. In other words, the operators and functions of a class T are *inherited* by all types in subclasses of T:

```
> :type (+) 5.0 2.0
(+) 5.0 2.0 :: (Fractional t) ⇒ t
```

Since we are now adding together numbers whose type is in the class **Fractional**, the resulting type belongs to the class **Fractional**. The fact that (+) can be used for summing up values of different types is an example of *overloading*: the same symbol (+ in this case) has different meanings (integral or fractional summation) depending on the type of its operands (or, more precisely, depending on the class to which the type of its operands belongs). In fact, type classes have been added to Haskell primarily as a tool for enabling operator overloading and have subsequently become an ubiquitous feature across the whole Haskell standard library.

We can also try a slight modification of this example to witness another interesting phenomenon:

```
> :type (+) 5 2.0
(+) 5 2.0 :: (Fractional t) ⇒ t
```

Here we are adding together two numbers of apparently different types. Indeed `5` is of type `(Num t) ⇒ t` and `2.0` is of type `(Fractional t) ⇒ t`. Also recall that (+) has type `(Num a) ⇒ a → a → a`, namely (+) accepts two arguments *of the same type* and produces a result whose type is the same as that of the arguments. In this example the arguments do *not* have the same type, and yet `GHCi` states that this is a well-typed expression. The reason lies in the fact that the type class **Fractional** is a subclass of **Num**. When `GHCi` tries to enforce that the two arguments must have the same type, but one belongs to the class **Num** and the other belongs to the class **Fractional**, it figures that if it forces the type of the first argument to belong to the class **Fractional** as well, then everything works fine. Observe that, had `GHCi` decided to go the other way around, by forcing `2.0` to have a type in the **Num** class, then some information could have been lost because in **Num** there are type classes (such as **Integral**) which do not admit fractional parts in numbers. The lesson here is that when two types must be the same, but they belong to different classes, then either there is no subclass relation between the classes, and then a type error occurs, or there is a subclass relation between the classes, and the subclass “wins” over the superclass.

While function applications account for most of the type errors, all the other syntactic constructs of the Haskell language do have type constraints that `GHCi` systematically verifies. So far, the only construct we know about is the conditional expression

if E_1 **then** E_2 **else** E_3

In an expression like this one, Haskell mandates two constraints: (1) the type of E_1 must be **Bool** and (2) E_2 and E_3 must have the same type. If either of these conditions is not met, a type error occurs:

```
> if 1 then 0 else 1
<interactive>:1:3:
    No instance for (Num Bool)
      arising from the literal ‘1’ at <interactive>:1:3
    Possible fix: add an instance declaration for (Num Bool)
    In the expression: 1
    In the expression: if 1 then 0 else 1
    In the definition of ‘it’: it = if 1 then 0 else 1
> if True then 1 else False
<interactive>:1:13:
    No instance for (Num Bool)
```

```
      arising from the literal ‘1’ at <interactive>:1:13
    Possible fix: add an instance declaration for (Num Bool)
    In the expression: 1
    In the expression: if True then 1 else False
    In the definition of ‘it’: it = if True then 1 else False
```

If the types of the expression in the two branches differ only because of the class they belong to, and there is a relation between the classes, then the whole expression is well typed and the subclass wins as for applications:

```
> :type if True then 1 else 2.0
if True then 1 else 2.0 :: (Fractional t) ⇒ t
```

3.7 Type annotations

Occasionally it is useful for the programmer to explicitly specify the type of an expression. This can be accomplished by writing the expressions followed by `::` followed by the intended type:

```
> :type 1 :: Int
1 :: Int :: Int
```

Observe that the type of `1` has been coerced to **`Int`**, as requested by the programmer.

These annotations can be used for retrieving the limits of bounded types such as **`Int`**, through the overloaded **`minBound`** and **`maxBound`** constants:[4]

```
> minBound :: Int
-2147483648
> maxBound :: Int
2147483647
```

3.8 Type conversions

We have seen that number literals are *overloaded*, meaning that they can be given different types depending on the context in which they occur. For example, the literal `1` can be interpreted as the integer number 1 or as the floating-point number 1.0. The fact that number literals are overloaded should not be confused with the implicit conversion (or promotion) rules implemented in other programming languages. As a clarifying example, consider the Haskell expression

```
> :type 1 + 2.5
1 + 2.5 :: (Fractional t) ⇒ t
```

which is well-typed and has type `(`**`Fractional`**` t) ⇒ t`. At first sight it might seem as if the integer number `1` is *promoted* to a fractional type, so that the addition is performed on operands with the same type. This is exactly what happens in other programming languages, including C, C++, C#, and Java. The above expression is well typed in all these languages, but the point is that `1` is considered an integer literal and `2.5` is considered

[4]The exact bounds may vary depending on the platform `GHCi` is running on, since Haskell does not mandate a precise size for value of type **`Int`**, unlike other languages such as Java.

a fractional literal and the language allows a silent promotion from integer values to fractional values. In Haskell there is no such promotion, and the above expression is well typed solely because 1 is not given an integer type in the first place, but a generic numeric type.

The difference between Haskell and the other programming languages can be appreciated by considering the following variation of the above expression:

```
> :type (1 :: Int) + 2.5
<interactive>:1:13:
    No instance for (Fractional Int)
      arising from the literal ‘2.5’ at <interactive>:1:13-15
    Possible fix: add an instance declaration for (Fractional Int)
    In the second argument of ‘(+)’, namely ‘2.5’
    In the expression: (1 :: Int) + 2.5
```

where we have explicitly coerced the type of literal 1 to **Int**. In this case GHCi complains because **Int**, the type of the left operand of +, is not an instance of the **Fractional** type class.

In order to use 1 :: **Int** and, more generally, a value of type **Int**, in a context where a fractional value is expected, the programmer must explicitly convert the integer number with the **fromIntegral** function:

```
> :type fromIntegral (1 :: Int) + 2.5
fromIntegral (1 :: Int) + 2.5 :: (Fractional b) ⇒ b
```

(recall that function application has precedence over any other operator, hence **fromIntegral** (1 :: **Int**) really means (**fromIntegral** (1 :: **Int**)) + 2.5).

Observe that **fromIntegral** is not simply a conveersion from **Int** to **Float**. Its type reveals that it can lift any value of any type that belongs to the class **Integral** to the corresponding value of any type that belongs to the class **Num**:

```
> :type fromIntegral
fromIntegral :: (Integral a, Num b) ⇒ a → b
```

In a sense, **fromIntegral** is an example of function that is overloaded not only in the type of its argument, but also in the type of the result! This is very convenient for the programmer as in practice there is only one name to remember, **fromIntegral**, that works equally well for a large class of conversions (indeed Haskell provides quite a lot of basic types that are instances of **Integral**, although we will work only with **Int**).

The opposite conversion, from fractional types to integral ones, can be performed in many ways, according to whether truncation or rounding are desired. For this Haskell provides two conversion functions called **truncate** and **round**:

```
> pi + 0.5
3.641592653589793
> truncate (pi + 0.5)
3
> round (pi + 0.5)
4
```

Once again, the types of **truncate** and **round** are more generic than the corresponding conversions in other programming languages:

```
> :type truncate
truncate :: (RealFrac a, Integral b) ⇒ a → b
> :type round
round :: (RealFrac a, Integral b) ⇒ a → b
```

where the type class `RealFrac` includes those types that are instance of the `Fractional` class.

3.9 Summary of Haskell type classes

We conclude this chapter with a diagram of the Haskell type classes we will be using in this course. The diagram (Table 3.1) is organized as a hierarchy of type classes. Each class shows the methods it provides along with their types. In these types it is implicitly assumed that the type being instance of the class is `a`. Arrows denote derivations between classes.

Note that this diagram shows only an approximation of the whole hierarchy of type classes defined in the Haskell standard library, and that even for the classes shown in the diagram it may be the case that not all methods are listed. The interested reader should refer to the Haskell reference manual for complete information.

3.10 Exercises

Exercise 3.1. Infer the type of the following expressions and use `GHCi` to check that your answers are correct:

1. `1 + 2 * 3`
2. `1 + 2 ** 3`
3. `1 + 2 ^ 3`
4. `1 + div 2 3`
5. `1 + (/) 2 3`
6. `1 + sin pi`
7. `pi / 2`
8. `if pi > 3 then 1 else (pi / 2)`
9. `if pi > 3 then 1 else (2 :: Int)`
10. `negate (if True then 1 else 2.0)`

Exercise 3.2. Explain the difference between `(^)` and `(**)` by looking at their types.

Exercise 3.3. Which of the following expressions are well-typed?

1. `div 2.0 1.0`
2. `(/) pi 3`
3. `if 2 < 3 then pi + 1 else False`
4. `(/) 2 3 4`
5. `negate pi`

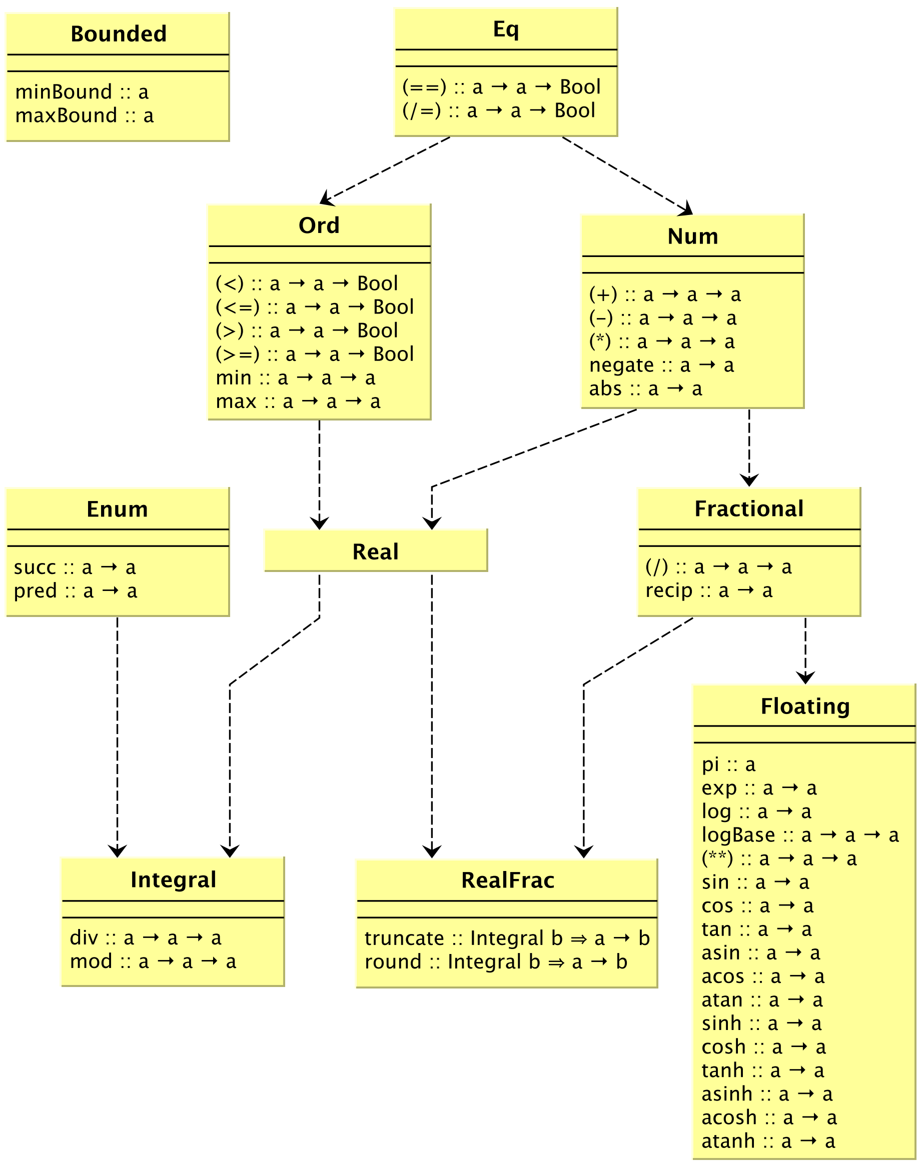

Table 3.1: Summary of Haskell type classes.

6. `abs (2 - 3)`
7. `abs 2 - 3`
8. `negate (-) 2 3`
9. `True :: Float`
10. `(1 :: Float) + (2 :: Int)`

Exercise 3.4. Add appropriate conversions so that the following expressions are well typed.

1. `mod 2 pi`
2. `sin (2 :: Int)`
3. `(2.5 :: Float) + (mod 3 2)`
4. `2 ^ pi`

Chapter 4

Definitions

Giving names to things is one of the most entertaining activities that human beings do. It is natural to give names to relevant results or values that occur often in a program. This practice is good for documentation purposes and it is also a way of saving expensive computations, by giving a name to the value they produce and repeatedly using the name as many times as necessary, whereas the expensive computation is performed only once.

A *binding* is an association between a name and a value. For instance, the fact that **pi** stands for (an approximation of) the constant π means that in the Haskell standard library there is a binding associating the name **pi** to its value. Similarly, there are bindings associating function names such as **div**, **negate**, and **sin** to function definitions.

4.1 Global definitions

The simplest way of defining a new binding is by means of a global (or top-level) definition, such as the following:

```
zero :: Int
zero = 0
```

Here we have defined a binding associating the name **zero** with the value 0. The binding is made of two separate lines: the first one gives **zero**, the name being defined, its type; the second line provides the actual value for **zero**.

The names to be used in definitions must adhere to the following lexical convention: a name must begin with a lowercase letter or the underscore symbol _ and is followed by zero or more of the following characters: lower- and uppercase letters, numbers, the underscore symbol _, the single quote '. If the name begins with an underscore symbol, it must be followed by at least another symbol.

The definition above is *not* an expression, hence it cannot be "evaluated" by GHCi. Rather, one writes a sequence of definitions in a so-called *Haskell script file*, which is nothing but a text file having extension .hs, then the script is loaded into GHCi and from that time on it is possible to evaluate expressions that make use the the definitions in the script. Assuming that the definition of **zero** above is included in a script called **zero**.hs, we can load it as follows:

```
> :load zero.hs
[1 of 1] Compiling Main             ( zero.hs, interpreted )
Ok, modules loaded: Main.
```

Here `GHCi` informs us that `zero.hs` has been loaded. By default, when a script is loaded, all the definitions it contains are included in a module called `Main`. We will see how to define a module in Chapter 17. The name `zero` stands for the value `0` of type `Int`, just as we have requested:

```
> zero + 1
1
> :type zero
zero :: Int
```

As a matter of facts the type declaration for a binding is optional, since Haskell is able of inferring the type by itself. However, it is generally good practice to explicitly provide the type for a binding, since this simplifies the understanding of the program and is a good form of documentation. In particular, when designing complex functions it is recommended starting off by writing their type.

The syntax `zero = 0` enforces the idea that `zero` and `0` are the same thing, and that the name `zero` can be used wherever `0` can (up to the type explicitly assigned to `zero`). An expression where the name `zero` occurs is evaluated by expanding `zero` with the associated value, thus:

$$1 + \texttt{zero} \Longrightarrow 1 + (0 :: \texttt{Int}) \Longrightarrow 1$$

This form of *equational reasoning* (i.e., substituting names with definitions) will become even more important with functions in later chapters.

Note that, because of the declaration `zero :: Int`, the expression `1 + zero` has type `Int`:

```
> :type 1 + zero
1 + zero :: Int
```

It is important not to confuse the symbol = in a definition with an *assignment operator*: the definition of `zero` might be misinterpreted as follows: first we declare a variable called `zero` of type `Int`; then, we initialize the variable `zero` with the value `0`. In functional languages, bindings are permanent initializations: once the value associated with a name is defined, it cannot be changed and there is no "assignment operator". By the way, multiple declarations of the same name in the same Haskell script are forbidden, even if the right hand side of each definition is exactly the same.

The order of definitions withing a Haskell script is irrelevant. The programmer does not need to care of the dependencies among the definitions, as Haskell can figure them out by itself. For example, the script

```
result :: Float
result = tan a

a :: Float
a = pi / 6
```

and the script

```
a :: Float
a = pi / 6

result :: Float
result = tan a
```

are completely equivalent.

4.2 Local definitions

Very often it is convenient to provide a name to an intermediate value, which is only used to determine the final result of a computation. In these cases using global definitions is excessive, as we risk to clutter the script with lots of definitions whose logical scope is actually very limited. Suppose, for example, that the name ‘a’ in the scripts at the end of the previous section is only useful for the definition of `result`, and nowhere else. It would be preferable to define a locally, only to determine the value of `result`. This can be accomplished by means of a *local definition*:

```
result :: Float
result = let a = pi / 6 in tan a
```

A local definition is made of the keyword **`let`** followed by one or more definitions followed by the keyword **`in`** followed by an expression. A local definition

`let` x = E_1 **`in`** E_2

is itself an expression whose value is that of E_2. However, in order to determine the value of E_2 it may be necessary to know the value of x, which is provided by the definition $x = E_1$.

In the example above we have not bothered giving a type to the local name a, since Haskell can infer it. In case we want to be explicit about the type of local names we can use the same syntax as for global definitions:

```
result :: Float
result = let
           a :: Float
           a = pi / 6
         in tan a
```

Since a local definition is itself an expression, it is possible to define multiple bindings by means of several **`let-in`** one after the other

```
result :: Float
result = let a = pi / 6 in
         let x = cos a in
         let y = sin a in
           y / x
```

as well as defining all of them in a single **`let-in`** and have Haskell figure out their dependencies

```
result :: Float
result = let x = cos a
             y = sin a
             a = pi / 6
         in
           y / x
```

In this last case, the indentation of the various definitions is important, because it lets Haskell understand where the definitions end and where the rest of the construct begins. In case a more liberal formatting is preferred (for example because the Haskell code is generated automatically by some other program), it is possible to group the definitions within curly braces and to separate them by means of semicolons:

```
result :: Float
result = let { x = cos a; y = sin a; a = pi / 6 } in y / x
```

4.3 Attaching local definitions to declarations

The **let-in** construct allows the programmer to build an expression that contains local definitions. Haskell also provides a similar construct for attaching local definitions to another declaration. For example, the last example in the previous section can be rewritten as follows:

```
result :: Float
result = y / x
    where
      x = cos a
      y = sin a
      a = pi / 6
```

This way we let the name `result` and its definition stay close together, and postpone all the auxiliary definitions after the keyword **where**.

4.4 Exercises

Exercise 4.1. Which of the following names can be used in definitions?

1. `x`
2. `3x`
3. `A`
4. `x’y`
5. `x’`
6. `’x`
7. `x3’`
8. `xA`
9. `x_3`
10. `_x`

Exercise 4.2. Determine the value of the following expressions:

1. `let x = pi / 2 in sin x ** 2`
2. `let x = 1 in if x > 0 then x else (negate x)`
3. `if True then let x = 2 in x ^ 4 else 0`
4. `negate (let { x = 1; y = 2 } in x + y)`
5. `let x = 1 in let y = 2 in x + y`
6. `let x = 1 in let x = 2 in x`
7. `let x = 1 in let y = 2 in let x = 0 in x + y`

8. `if let x = True in x then let x = 1 in x else let x = 2 in x`

Exercise 4.3. Write an expression that reduces to 2^{16} using only the literal `2`, no more than 4 occurrences of the operator `*`, and no other literal or operator or function.

Part II

Functions

Chapter 5

Defining functions

Now that we know about representing numbers and boolean values in a Haskell program, we need some way of transforming them. In a functional programming language this is accomplished by defining *functions*.

5.1 Transforming values

The successor function that takes an integer value and increments it by one can be defined thus:

```
successor :: Int → Int
successor n = 1 + n
```

Just as we have seen in Chapter 4, the definition of `successor` is split into two parts. The first line gives the type of `successor`. In this case, the type `Int → Int` denotes that `successor` is a function that accepts an integer value and returns an integer value. The second line defines the actual behavior of `successor`: it states that the function `successor` applied to an argument `n` is equal to `1 + n`. In this definition, the name `n` on the left hand side of `=` stands for the *formal argument* of the function, which is to be replaced by an *actual argument* when the function is applied. The expression on the right hand side of `=` is often called the *body* of the `successor` function. The name of argument, `n` in this example, can be freely chosen by the programmer, but it must be an identifier following the same conventions for names that have been described in Section 4.1.

We may now compute the successor of the number `2` thus:

```
> successor 2
3
```

where we write the name of the function to be applied, `successor`, next to the actual argument to which it is applied, `2`.

We can think of the definition of `successor` as of a rewriting rule saying that the application of `successor` to an argument `n` is rewritten to `1 + n`. Thus, the computation of `successor 2` proceeds as follows:

```
  successor 2 ⟹ 1 + 2 ⟹ 3
```

where, during the first rewriting, the argument `n` in the body of the function is replaced by the actual argument `2`. The second rewriting step follows by the intuitive meaning of the `+` operator. Another way of looking at the definition of `successor` is as an *equation* saying that `successor n` and `1 + n` are *the same thing*. For these reasons, we will often refer to the equations or rules defining a function.

Recall that function application has the highest precedence over all the other operators. Hence, when computing the successor of a compound expression, one should not forget to properly group the expression within parentheses. Compare

```
> successor (2 * 3)
7
```

and

```
> successor 2 * 3
9
```

where in the last example the lack of parentheses gives rise to the following computation

```
successor 2 * 3 ⟹ (1 + 2) * 3 ⟹ 3 * 3 ⟹ 9
```

Example 5.1 (reciprocal of a number). The standard Haskell library provides a function called **recip** to compute the reciprocal of a number. Namely, **recip** applied to a number n returns $\frac{1}{n}$. We can define **recip** as follows:

```
recip :: Float → Float
recip n = 1 / n
```

but this makes **recip** only applicable to floating-point numbers. Since the (/) operator can be applied to values whose type is an instance of the **Fractional** type class, we can generalize **recip** like this:

```
recip :: (Fractional a) ⇒ a → a
recip n = 1 / n
```

■

Example 5.2. Suppose we need to write a function **even** to test whether an integer number is even or not. A function that returns a boolean value is sometimes called *predicate*. We could implement **even** like this:

```
even :: Int → Bool
even n = if n ‘mod‘ 2 == 0 then True else False
```

Basically, **even** accepts an argument `n` and checks whether the remainder of `n` divided by 2 is zero. According to the result, **even** reduces to either **True** or **False**.

Observe however that `n ‘mod‘ 2 == 0` is itself a boolean expression, and that its value coincide with the value returned by **even**. For these reasons, we can simplify **even** like this:

```
even :: Int → Bool
even n = n ‘mod‘ 2 == 0
```

Finally, requiring that the argument of **even** must be an integer number is a useless restriction. We can turn **even** to an overloaded function that seamlessly works with any integral type, thus:

```
even :: (Integral a) ⇒ a → Bool
even n = n ‘mod‘ 2 == 0
```

■

Example 5.3 (Stirling's formula). Stirling's formula provides an approximation for large factorials. Recall that the factorial of a natural number n is denoted by $n!$ and is defined as the product of all numbers between 1 and n. Stirling's formula is written as

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

We can define a Haskell function for computing Stirling's approximation thus:

```
stirling :: Int → Float
stirling n = sqrt (2 * pi * n') * ((n' / exp 1) ** n')
  where
    n' :: Float
    n' = fromIntegral n
```

where we locally define a name `n'` equivalent to `n`, except that `n'` has type `n`. This is convenient since `n'` is needed multiple times in the formula. ■

5.2 Guards

Suppose we want to implement a function to compute the absolute value of an integer number. The idea is to write a function `absolute` accepting a number `n` and comparing `n` against `0`: if `n` is negative, then we return `-n`; if `n` is non-negative, we return it unchanged. We can write `absolute` using a conditional expression, thus:

```
absolute :: Int → Int
absolute n = if n >= 0 then n else negate n
```

Basically `absolute` has two different behaviors, according to whether `n` is negative or not.

Haskell gives us a more declarative style of writing functions like `absolute`, by separating the different behaviors and choosing among them by means of *guards*:

```
absolute :: Int → Int
absolute n | n >= 0 = n
           | n < 0 = negate n
```

Each guard is an expression of type **`Bool`** preceeded by `|` and followed by `=`. Intuitively, this definition means that `absolute n` is equal to `n` when `n >= 0`, and it is equal to **`negate`** `n` when `n < 0`. A few tests suffice to see that `absolute` is working properly:

```
> absolute 5
5
> absolute (-5)
5
```

Yet this version of `absolute` is slightly unsatisfactory, if compared to the one above using the conditional expression, in that one has the feeling that the second guard states something obvious. Since guards are tried from top to bottom, if the guard `n >= 0` fails then obviously it must be that `n < 0` succeeds, but it would be better to avoid writing the guard altogether. Haskell defines a special guard expression, called **`otherwise`**, that serves perfectly in this case:

```
absolute :: Int → Int
absolute n | n >= 0 = n
           | otherwise = negate n
```

In fact, **otherwise** is nothing but an alias for the boolean value **True**:

```
> :type otherwise
otherwise :: Bool
> otherwise
True
```

Example 5.4 (sign of a number). The library function **signum** takes a number and returns −1, 0, or 1 according to whether the number is negative, zero, or positive respectively. For integer numbers, **signum** can be defined thus:

```
signum :: Int → Int
signum n | n < 0 = -1
         | n > 0 = 1
         | otherwise = 0
```

The actual library function is overloaded and can be applied to any number, not just integers. ■

5.3 First steps with pattern matching

Suppose we need a function `safeRecip` to compute the reciprocal of a number that returns 0 if the number is itself 0. We can take full advantage of guarded definitions and write it like this:

```
safeRecip :: (Fractional a) ⇒ a → a
safeRecip n | n == 0 = 0
            | otherwise = 1 / n
```

Observe that in the first guard of `safeRecip` we are comparing the argument `n` against a specific value, `0` in this case. Haskell gives us yet another way of writing this function even more declaratively:

```
safeRecip :: (Fractional a) ⇒ a → a
safeRecip 0 = 0
safeRecip n = 1 / n
```

In this version, `safeRecip` is defined by means of two separate rules. Each rule begins with the name of the function, followed by a *pattern*. The pattern specifies a structural template that the argument must satisfy in order for the rule to be applicable. In this example, the first rule says that `safeRecip` returns `0`, but only when it is applied to the argument `0`. The second rule, which is tried only if the first one fails, says that `safeRecip` returns `1 / n` in all the other cases. Technically speaking, the pattern `0` in the first rule matches only the value `0`, whereas the pattern `n` matches any (other) value:

```
> safeRecip 0
0.0
> safeRecip 2
0.5
> safeRecip 3
0.3333333333333333
```

In this style of definition the order of the rules does matter, since rules are tried from top to bottom, just like guards. If we define a similar `safeRecip` function with the two rules swapped, as in

```
safeRecip :: (Fractional a) ⇒ a → a
safeRecip n = 1 / n
safeRecip 0 = 0
```

then the first rule – with pattern `n` – is always applicable and “hides” the second one. In many cases GHCi spots this sort of mistakes and warns the programmer with a message like the following:

```
   Warning: Pattern match(es) are overlapped
            In the definition of ‘safeRecip’: safeRecip 0 = ...
```

In general, one should carfeully think of the order in which several equations for the same function are given, especially for multi-argument functions that we will explore shortly. Indeed, the last version of `safeRecip` is not “safe”, at least in the sense that it does not return 0 if applied to 0:

```
> safeRecip 0
Infinity
```

Example 5.5 (boolean negation). The library function **not** negates a boolean value. It can be defined using a conditional expression thus:

```
not :: Bool → Bool
not b = if b then False else True
```

or more declaratively using pattern matching and two distinct equations thus:

```
not :: Bool → Bool
not True = False
not False = True
```

■

5.4 Multi-argument functions

Defining multi-argument functions is simply a matter of generalizing the syntax we have seen for single argument functions. When writing the equations, we can use as many patterns as necessary on the left hand side of =. For example, we can compute the minimum and maximum of two integer numbers as follows:

```
intMin :: Int → Int → Int
intMin m n | m < n = m
           | otherwise = n

intMax :: Int → Int → Int
intMax m n | m > n = m
           | otherwise = n
```

Observe the type of `intMin` and `intMax`: there are as many arrows as the number of arguments of the function. The type preceding each arrow is the type of the corresponding argument. The last type is the type of the result.

When applying a multi-argument function, just write the arguments next to the function name separated by spaces:

```
> intMin 2 3
2
> intMax 2 3
3
```

As usual, parentheses must be used for complex arguments. Compare

```
> intMin 2 1 + 2
3
```

and

```
> intMin 2 (1 + 2)
2
```

Recall that each argument is specified by means of a pattern. In `intMin` and `intMax` the patterns corresponding to the two arguments are simply `m` and `n`. Consider the following function `divides`, which verifies whether the first argument divides the second one:

```
divides :: Int → Int → Bool
divides m n = m /= 0 && n ‘mod‘ m == 0
```

The idea is that if the first argument of `divides` is `0`, the result is **`False`** regardless of the value of the second argument. However, we must check `m /= 0` for otherwise we would get a "division by zero" error when computing `n ‘mod‘ m`. An alternative, more declarative way of implementing `divides` using patterns is the following:

```
divides :: Int → Int → Bool
divides 0 n = False
divides m n = n ‘mod‘ m == 0
```

The first equation states that `divides` returns **`False`** if the first argument is `0` and regardless of the value of the second argument. The second equation, which is used only if the first one fails, namely if `m` is different from `0`, handles the interesting case. A few tests suffice to convince ourselves that `divides` works properly:

```
> 2 ‘divides‘ 3
False
> 0 ‘divides‘ 3
False
> 2 ‘divides‘ 4
True
```

When writing multi-argument functions with multiple equations, one should make sure that each equation has the same number of arguments and that each equation is consistent with the type declaration for the function. In addition, Haskell imposes a *linearity constraint* on every equation: there must not be a pattern variable that occurs in two or more places on the left hand side of an equation. So for instance

```
  f n n = ...
```

is an illegal equation, since the pattern variable `n` occurs twice. If one needs to specify that two or more arguments must be equal, it is possible to do so by means of guards. Hence the illegal equation above can be corrected thus:

```
  f m n | m == n = ...
```

As a final consideration, note that in the first rule in the definition of `divides` the name of the second argument `n` is never used on the right hand side of `=`. We can spare one name and obtain an equivalent definition by means of the pattern `_`, which matches any value:

```
divides :: Int → Int → Bool
divides 0 _ = False
divides m n = n ‘mod‘ m == 0
```

The reader is invited to try out both versions of `divide` to convince herself that they are equivalent.

Example 5.6 (minimum and maximum). The library functions **`min`** and **`max`** are generalizations of `intMin` and `intMax` seen above that work with values of arbitrary type, so long as this type is an instance of the **`ord`** class. They are defined as follows:

```
min :: (Ord a) ⇒ a → a → a
min m n | m < n = m
        | otherwise = n

max :: (Ord a) ⇒ a → a → a
max m n | m > n = m
        | otherwise = n
```

Observe that they differ from `intMin` and `intMax` just because of the type, which makes them overloaded. ■

5.5 Exercises

Exercise 5.1. Determine which of the following function definitions are well typed and, for those that are well typed, determine their type.

1. `f x = sin (fromIntegral x)`
2. `f x = x ** 1`
3. `f b = if b then sin else False`
4. `f x = sin sin x`
5. `f x = (sin sin) x`
6. `f x = sin (sin x)`
7. `f x y = x + y + 1.5`
8. `f x y z = if x then y else z`
9. `f x y = if x then y + 1 else True`
10. `f x y u v = v + (if x < y then 0 else u)`

Exercise 5.2. Define a function `sign` that, applied to an integer number n, returns -1, 0, or 1 depending on the sign of n (negative, zero, or positive).

Exercise 5.3. Define two conversion functions `boolToInt` from boolean values to integer numbers and `intToBool` from integer numbers to boolean values in the spirit of the C language, where an integer number other than zero is considered "true", and zero is considered "false".

Exercise 5.4. Define a function `nearest` from floating-point numbers to integer numbers that rounds its argument to the *nearest* integer. Make sure that the function works properly also for negative numbers. Check your implementation against the library function **`round`**.

Exercise 5.5. Define a function `fractional` that returns the fractional part of a fractional number. For example `fractional 2.25` should evaluate to `0.25` and `fractional 8` should evaluate to `0.0`.

Exercise 5.6. Define two conversion functions `degrees` and `radians` to respectively convert angles from degress to radians and from radians to degrees. Define a few variants of the trigonometric library functions in Table 2.2 that accept and return degrees instead of radians.

Exercise 5.7. A year is *leap* if it can be divided by 4 but not by 100, or if it can be divided by 400. For example 1984 is leap, 1900 is not leap, and 2000 is leap. Define a predicate `leap` that evaluates to **`True`** when applied to a leap year and to **`False`** otherwise.

Exercise 5.8. Write Haskell functions corresponding to the following mathematical functions:

$$f(a,b,x) = ax + b \tag{5.1}$$

$$f(a,b,c,x) = ax^2 + bx + c \tag{5.2}$$

$$f(n,x) = \sin^n x + \cos^n x \tag{5.3}$$

$$f(r,s) = \frac{\pi^2(r+s)(r-s)^2}{4} \tag{5.4}$$

$$f(x,y) = \sqrt[x]{y} \tag{5.5}$$

Exercise 5.9. Define a function that, when applied to two floating-point numbers representing the real and imaginary part of a complex number c, evalutates to the modulus of c.

Chapter 6

Recursion

Every nontrivial program typically performs a number of operations that depends on the "size of the input" to the program. For this reason, most programs are *iterative* in nature, namely they are built on fragments of code that are executed over and over again. In a functional programming language such as Haskell, iteration is implemented by means of recursive definitions.

6.1 Recursive definitions

Consider the problem of determining the volume of a square-based pyramid made of cubic blocks. Suppose that the base of the pyramid measures n blocks, that the level immediately on top of it measures $n - 1$ blocks, and so on to the tip of pyramid, which is made of 1 block only. Then, the base of the pyramid is made of n^2 blocks, the level immediately on top of it is made of $(n - 1)^2$ blocks, and so on. The total volume of the pyramid is

$$n^2 + (n - 1)^2 + (n - 2)^2 + \cdots + 3^2 + 2^2 + 1$$

blocks.

One way of computing this sum is by means of an *iterative algorithm* that takes as input a natural number n and sums up the square of each number i between 1 and n.[1] Given that all we have in Haskell is functions and function application, it is not immediately clear how to implement this algorithm though.

Let us write $P(n)$ for the volume of the pyramid whose base is n blocks wide. The first key observation is that the pyramid made of 1 level is so simple that its volume can be computed immediately: if it's made of 1 level, then it must be made of just 1 block, hence

$$P(1) = 1 \tag{6.1}$$

The second key observation is that every "big pyramid" (having more than one level) is composed of a base and a *smaller* pyramid on top of it. In particular, if the base is n blocks wide, then the pyramid on top of it is $n - 1$ blocks wide. Thus the following equation holds:

$$P(n) = n^2 + P(n - 1) \tag{6.2}$$

namely the volume of a pyramid with base n is equal to n^2 (the overall number of blocks in the base) plus the volume of the pyramid on top of the base, which is given by $P(n - 1)$.

[1]Another way is to compute $\frac{n(n+1)(2n+1)}{6}$ but that would be considered as cheating in this chapter!

By unfolding $P(n)$ we obtain

$$\begin{aligned} P(n) &= n^2 + P(n-1) \\ &= n^2 + (n-1)^2 + P(n-2) \\ &= n^2 + (n-1)^2 + (n-2)^2 + P(n-3) \\ &= \cdots \end{aligned}$$

until we eventually reach $P(1)$ which is trivially equal to 1, by the reasoning above.

We have just given a *recursive* method for computing the volume of a pyramid. The method is called “recursive” because, roughly speaking, “it is defined in terms of itself”. Literally this does not mean anything, but if we observe equation 6.2 carefully we see that $P(n)$ (the volume of the pyrammid with base n) is indeed defined in terms of $P(n-1)$ (the volume of the pyramid with base $n-1$). Thus we have provided a solution of the problem “for input n” in terms of the solution of the *same* problem but for a *smaller* input. Since we also have a simple case (equation 6.1) that is eventually necessary and that does not involve any recursion ($P(1)$ is just 1), this recursive definition is *well founded*.

In Haskell we would implement P as a function `pyramid` as follows:

```
pyramid :: Int → Int
pyramid 1 = 1
pyramid n = n ^ 2 + pyramid (n - 1)
```

Observe that the definition of `pyramid` is made of two equations: the first one takes care of the simple case, whereas the second equation defines the value of `pyramid n` in terms of the value of `pyramid (n - 1)`.

A few tests show that `pyramid` does work correctly:

```
> pyramid 10
385
> pyramid 20
2870
> pyramid 1
1
```

Example 6.1 (factorial). The factorial of a natural number n, denoted by $n!$, is the product of all number numbers between 1 and n, included. Namely

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

where $0! = 1$ by convention. The factorial can be implemented recursively pretty much as the volume of the pyramid:

```
fact :: Int → Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Observe that the structure of the code is the same as that of `pyramid`. We have only changed the base case and the way the result is obtained in the inductive case, once the recursion has been completed. ■

Example 6.2 (Fibonacci numbers). The sequence of Fibonacci numbers begins like this:[2]

```
  0 1 1 2 3 5 8 13 21 34 55 89 ···
```

[2]In some presentations the sequence does not include the initial `0`.

The sequence begins with 0 and 1, and every following number is the sum of the previous two. We can implement a function `fibo` that computes the n-th Fibonacci number as follows:

```
fibo :: Int → Int
fibo 0 = 0
fibo 1 = 1
fibo n = fibo (n - 1) + fibo (n - 2)
```

Indeed `fibo` n (the n-th Fibonacci number) is the sum of the $(n-1)$-th and of the $(n-2)$-th Fibonacci numbers. For example, when computing `fibo 2` we reduce to computing both `fibo 0` and `fibo 1`. By having the base case `fibo 0 = 0` we make sure that `fibo 2 = 1`, as required. ∎

Example 6.3 (power). The n-th power of an integer number a can be inductively defined as follows:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n/2} \times a^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \\ a \times a^{n/2} \times a^{n/2} & \text{otherwise} \end{cases}$$

The advantage of this inductive definition is that the computation of a^n involves only $\log_2 n$ multiplications, as opposed to n with the trivial definition of a^n as "a multiplied by itself n times". Along the lines of this definition, we can define a Haskell function for computing a^n thus:

```
power :: Int → Int → Int
power _ 0 = 1
power a n | even n = a' * a'
          | otherwise = a * a' * a'
    where
      a' = power a (n `div` 2)
```

∎

6.2 Iteration, state and recursion

Iterative problems rely on a notion of modifiable state. For example one formulation of Euclid's algorithm for computing the greatest common divisor of two positive numbers m and n is the following:

1. if $m = 0$, then the greatest common divisor of m and n is n;
2. if $m < n$, then swap m and n;
3. replace m with $m - n$ and go back to step 1.

The algorithm keeps track of the state in two variables m and n. Before running the algorithm, the variables are initialized with two positive numbers. The fact that the algorithm modifies the state can be clearly seen in steps 2 and 3. In step 2, the contents of the two variables m and n are "swapped" if m turns out to be smaller than n. In step 3, the content of variable m is replaced by $m - n$. We can visualize the algorithm at work on the numbers $m = 6$ and $n = 15$ as a sequence of states, thus:

$$\begin{array}{c}\boxed{m \mid 6}\\ \boxed{n \mid 15}\end{array} \Longrightarrow \begin{array}{c}\boxed{m \mid 15}\\ \boxed{n \mid 6}\end{array} \Longrightarrow \begin{array}{c}\boxed{m \mid 9}\\ \boxed{n \mid 6}\end{array} \Longrightarrow \begin{array}{c}\boxed{m \mid 3}\\ \boxed{n \mid 6}\end{array} \Longrightarrow \begin{array}{c}\boxed{m \mid 6}\\ \boxed{n \mid 3}\end{array} \Longrightarrow \begin{array}{c}\boxed{m \mid 3}\\ \boxed{n \mid 3}\end{array} \Longrightarrow \begin{array}{c}\boxed{m \mid 0}\\ \boxed{n \mid 3}\end{array}$$

In a pure functional language such as Haskell it is not immediately obvious how to implement an algorithm like this one, where state modification is a key ingredient. The idea is to represent the state, the variables m and n in this case, as the arguments of a recursive function. Whenever the state must be changed, the function calls itself recursively with a new pair of arguments representing the updated state. With this insight, we can readily define a Haskell function `euclid` to compute the gratest common divisor of two integer numbers thus:

```
euclid :: Int → Int → Int
euclid 0 n = n
euclid m n | m < n = euclid n m
           | otherwise = euclid (m - n) n
```

The first equation for `euclid` m n deals with the case when m is zero, in which case the computation is over and the function immediately returns the value of n. The second equation is split in two subcases: if $m < n$, then we must update the state by swapping m and n and we do so by recursively applying `euclid` to n (first) and m (second); if $m \geq n$, then we must update m by replacing it with $m - n$ and we do so by recursively applying `euclid` to $m - n$ and n. Observe that in the last subcase n “remains unchanged”.

By evaluating `euclid 6 15` we obtain

```
> euclid 6 15
3
```

and by visualizing the sequence of reduction steps

```
euclid 6 15 ⟹ euclid 15 6 ⟹ euclid 9 6 ⟹ euclid 3 6
            ⟹ euclid 6 3 ⟹ euclid 3 3 ⟹ 0 3 ⟹ 3
```

we can appreciate the strict correspondence between the value of the arguments of the `euclid` function and the content of the cells m and n in the sequence of states above.

Sometimes an algorithm in iterative form may be far more efficient than a recursive one. For example, it is possible to demonstrate that the function `fibo` in Example 6.2 performs a number of reductions that is exponential in the index k of the Fibonacci number we want to compute. The intuition is that the evaluation of `fibo n` requires evaluating both `fibo (n - 1)` and `fibo (n - 2)`, but the evaluation of `fibo (n - 1)` also requires the evaluation of `fibo (n - 2)`. Hence, `fibo (n - 2)` is independently evaluated twice.

Nonetheless, it is not too hard to design an iterative algorithm that solves the same problem in linear time:

1. initialize m to 0 and n to 1;
2. if $k = 0$, return the value of m;
3. if $k = 1$, return the value of n;
4. simultaneously replace m with n and n with $m + n$;
5. decrement k by 1 and go back to step 2.

This algorithm assumes that the variable k is initialized with the index of the Fibonacci number one wants to compute. The two auxiliary variables m and n remember the last two

Fibonacci numbers that have been computed. For example, we can visualize the computation of the 7-th Fibonacci number:

$$
\begin{array}{|c|c|}\hline m & 0 \\ \hline n & 1 \\ \hline k & 7 \\ \hline \end{array}
\Longrightarrow
\begin{array}{|c|c|}\hline m & 1 \\ \hline n & 1 \\ \hline k & 6 \\ \hline \end{array}
\Longrightarrow
\begin{array}{|c|c|}\hline m & 1 \\ \hline n & 2 \\ \hline k & 5 \\ \hline \end{array}
\Longrightarrow
\begin{array}{|c|c|}\hline m & 2 \\ \hline n & 3 \\ \hline k & 4 \\ \hline \end{array}
\Longrightarrow
\begin{array}{|c|c|}\hline m & 3 \\ \hline n & 5 \\ \hline k & 3 \\ \hline \end{array}
\Longrightarrow
\begin{array}{|c|c|}\hline m & 5 \\ \hline n & 8 \\ \hline k & 2 \\ \hline \end{array}
\Longrightarrow
\begin{array}{|c|c|}\hline m & 8 \\ \hline n & 13 \\ \hline k & 1 \\ \hline \end{array}
$$

We can implement this algorithm in Haskell by defining an auxiliary recursive function `aux` with three parameters corresponding to the variables m, n, and k:

```
aux :: Int → Int → Int → Int
aux m _ 0 = m
aux _ n 1 = n
aux m n k = aux n (m + n) (k - 1)
```

Strictly speaking, this function does not compute the k-th Fibonacci number, because it also requires the initial values for m and n as it first two arguments. Thus, we can re-implement `fibo` as follows:

```
fibo :: Int → Int
fibo k = aux 0 1 k
```

where `fibo k` evaluates to `aux 0 1 k`. Observe that the initial value of the first two parameters of `aux` are exactly the values used for initializing the variables m and n in the algorithm above:

```
> fibo 7
13
```

As a final remark, since `aux` makes sense only within the context of `fibo`, it is cleaner to declare it locally:

```
fibo :: Int → Int
fibo k = aux 0 1 k
    where
      aux :: Int → Int → Int → Int
      aux m _ 0 = m
      aux _ n 1 = n
      aux m n k = aux n (m + n) (k - 1)
```

6.3 Generalization

The key idea of recursion is being able to solve a problem for some input using the solution of the same problem for a “smaller” input. The precise meaning of “smaller” usually depends on the problem. In all the examples that we have seen so far “smaller” means “being a smaller number”, but we will see that very different interpretations may be used. There are problems, however, for which there is no connection whatsoever between the solution for some input and the solution of the same problem for any smaller input.

As an example, consider the problem of determining whether an integer number n is *prime*. Recall that n is prime if its only divisors are 1 and n, and that by convention 1 is *not* prime. The fact that n is prime has nothing to do with the fact that any number smaller than n is or isn’t prime. So how can we set up a recursive algorithm for testing whether n is prime?

In this case it helps to *generalize* the problem. Let us define a predicate $D(n,k)$ such that $D(n,k)$ holds when there is no number between 2 and k that divides n. According to the definition of prime number, it is easy to see that n is prime if and only if $D(n,n-1)$ holds. The advantage of working with $D(n,k)$ instead of working directly with the property of being a prime number is that $D(n,k)$ can be expressed recursively. If $k = 1$, then $D(n,k)$ holds trivially, since there is no number between 2 and 1. We have just identified the base case! If $k > 1$, then $D(n,k)$ holds if k does not divide n and $D(n,k-1)$ holds. Thus, we can solve the problem "$D(n,k)$ holds" if we have a solution of the problem "$D(n,k-1)$ holds". This is the recursive case. With this insight we can implement `prime` with an auxililary function `aux` that solves the generalized problem, thus:

```
prime :: Int → Bool
prime n = n > 1 && aux (n - 1)
    where
      aux :: Int → Bool
      aux 1 = True
      aux k = n ‘mod‘ k /= 0 && aux (k - 1)
```

```
> prime 11
True
> prime 101
True
> prime 2
True
> prime 4
False
```

A few observations to conclude this example: first of all, just as we did with `fibo` above, we have defined `aux` locally to `prime`, since it only makes sense in this restricted scope; second, the test `n > 1` in the equation for `prime n` is fundamental for the correctness of this implementation because there is no prime number that is strictly less than 2; third, unlike $D(n,k)$ which is a predicate in two variables n and k, `aux` has just one argument. Being defined locally for `prime`, it is perfectly legal for `aux` to access `prime`'s argument `n`, which never changes through recursive applications.

6.4 Case study: combinations

Suppose we need to compute the number of m-combinations of n distinct elements, namely the number of subsets with m elements of a set S with n elements. This number is usually represented as

$$\binom{n}{m}$$

and it is well known, from combinatorial mathematics, that

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Given that we know how to compute $n!$ and $m!$, we could define a function `comb` as follows:

```
comb :: Int → Int → Int
comb n m = fact n / (fact m * fact (n - m))
```

However, we can also compute $\binom{n}{m}$ inductively by observing that:

- if $m = n$, then there is only one such subset, which is obtained by taking *all* the n elements in the set;
- if $m = 0$, then there is only one such subset, which is obtained by taking *no* elment from the set S;
- if $0 < m < n$, then we can build a subset of S with m elements in two ways: either we choose one of the n elements in S and put it in the subset, and then build a subset of $m - 1$ elements from a set with $n - 1$ elements, or we discard one of the n elements in S and then build a subset of m elements from a set with $n - 1$ elements.

We have just determined the following equation:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \text{ or } m = n \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{otherwise} \end{cases}$$

from which we derive the following alternative implementation of `comb`:

```
comb :: Int → Int → Int
comb n m | m == 0 || m == n = 1
         | otherwise = comb (n - 1) (m - 1) + comb (n - 1) m
```

6.5 Exercises

Exercise 6.1. Is it possible to fix the definition of `prime` and `aux` in Section 6.3 so that the test `n > 1` can be omitted?

Exercise 6.2. The definition of `prime` can be seen as another instance of iterative algorithm turned into a recursive one. Write down the corresponding iterative algorithm.

Exercise 6.3. Define a recursive function that, when applied to two integer numbers n and m, evaluates to n^m. Do not make use of any exponentiation operator. For simplicity, assume $m \geq 0$.

Exercise 6.4. Define a function that, when applied to an integer number $n \geq 0$, returns the number of 1's in the binary representation of n.

Exercise 6.5. A number is *perfect* if it is equal to the sum of all of its divisors excluding itself. Define a predicate `perfect` that evalutates to **`True`** if applied to a perfect number, and to **`False`** in all the other cases.

Exercise 6.6. Define a function to find the smallest prime number that is larger than or equal to a given number n.

Chapter 7

Polymorphic functions

Polymorphism, literally meaning “having many forms”, is the ability to write functions that seamlessly work when applied to arguments of different types.

7.1 Polymorphism as type abstraction

Perhaps the simplest Haskell function is the identity function: when applied to an argument, it spits out the argument unchanged. A possible implementation of the identity function is the following:

```
id :: Int → Int
id x = x
```

but this immediately raises the question: why should the argument be of type **Int**? If we need the identity function over floating-point numbers, then we’d better define a `idFloat` like this:

```
idFloat :: Float → Float
idFloat x = x
```

And what if we need to apply the identity function to a boolean value? Well, let’s define a `idBool` then:

```
idBool :: Bool → Bool
idBool x = x
```

And so on and so forth. Because of arrow types, there are an infinite number of identity functions that may be worth writing (and more types are going to be introduced in the next chapters). This way of writing code breaks one of the fundamental principles of software engineering: code reuse. We keep writing the same code for the identity function over and over again, just because the types of **id**, `idFloat`, and `idBool` are different.

The key observation is that the actual type of the argument of **id** does not matter, because **id** is never going to do anything with the argument. No function and no operator is applied to the argument in the body of **id**. Thus, it would be nice to be able to give **id** a type like this:

```
id :: Any → Any
```

where `Any` is an hypothetical type meaning, indeed, any type.

The problem of this approach is that `Any` would have to be compatible with any other type. For example, the expressions

```
id 1
```

and

```
id True
```

would all be well typed, but then also

```
if True then id 1 else id True
```

would, because the expressions `id 1` and `id True` in the two branches would both have type `Any`. This last example does clearly have a problem though, because `id 1` reduces to an integer number while `id True` reduces to a boolean value, while we know that the two branches of a conditional expression must reduce to values of the same type.

Giving `id` the type `Any → Any` means approximating too much the behavior of `id`. We loose the information that the value returned by `id` has the same type as the argument `id` is applied to. The idea to make the type of the argument of `id` *abstract* by giving it a *name*. Then, in the codomain of `id`'s type we use the same name indicating that the function returns a value whose type is the same as the type of its argument. The name we give to the type is called *type variable*, and `id` can be defined thus:

```
id :: a → a
id x = x
```

Here `a` is a type variable roughly meaning "any type", just like `Any`. However, we are saying that the value returned by `id` has the same type as the type of the argument. The arrow type `a → a` is said to be a *polymorphic type* because it contains a type variable. It reads "for every type `a`, `a → a`", thus if `id` is applied to an argument of type `Int`, then it returns a value of type `Int`; if it is applied to an argument of type `Float`, then it returns a value of type `Float`; if it is applied to an argument of type `Bool`, then it returns a value of type `Bool`; and so on and so forth. In other words, we have defined a single version of the identity function which has many different types such as `Int → Int` or `Float → Float` or `Bool → Bool` and so forth. Each of these types is a particular *instantiation* of the type `a → a`.

Example 7.1. Suppose we need two functions called `first` and `second` that take two arguments and return the first one or the second one, respectively. They can be defined thus:

```
first :: a → b → a
first x _ = x

second :: a → b → b
second _ x = x
```

Observe that both `first` and `second` have a polymorphic type which reflects the behavior of the function: both accept two arguments of type "`a`" and "`b`" respectively. However, `first` returns something of type "`a`" which is the type of the first argument, while `second` returns something of type "`b`" which is the type of the second argument. ■

7.2 Applying polymorphic functions

Polymorphic functions are applied just as any other function. For example, we can apply `id` like this:

```
> id 1
1
> id 2.0
2.0
> id True
True
```

If we ask GHCi about the type of **id** we obtain

```
> :type id
id :: a → a
```

as expected, but if we ask GHCi the type of the expressions above, we obtain instead

```
> :type id 1
id 1 :: (Num t) ⇒ t
> :type id 2.0
id 2.0 :: (Fractional t) ⇒ t
> :type id True
id True :: Bool
```

that is, GHCi *instantiates* the type of **id** depending on the type of the argument it is applied to. Let us look in detail the last example, where **id** is applied to the boolean value **True**. We know that the type of **id** is a → a and that the type of **True** is **Bool**. For **id True** to be well typed, the type in the domain of **id**, "a", must match **Bool**. This is obtained by replacing the "a" type variable with the type **Bool**, hence the type of **id** becomes **Bool → Bool**. As a consequence, the whole expression **id True** has type **Bool**.

Every occurrence of **id** is independent of the others, so it is typed independently. When writing **id** 1 GHCi figures that *this occurrence* of **id** has type

```
(Num t) ⇒ t → t
```

whereas when writing **id True** GHCi figures that *this occurrence* of **id** has type

```
Bool → Bool
```

Multiple occurrences of **id** are independent also within the same expression. For example, the expression

```
> if id True then id 1 else 2
1
> :type if id True then id 1 else 2
if id True then id 1 else 2 :: (Num t) ⇒ t
```

is well typed and **id** occurs twice in it, each occurrence being instantiated with a different type.

Example 7.2. In Chapter 2 we have introduced conditional expressions as a primitive Haskell feature. By now we can see that conditional expressions can be implemented with polymorphic functions and function application. In particular, let ifFun be the polymorphic function defined as follows:

```
ifFun :: Bool → a → a → a
ifFun True e _ = e
ifFun False _ e = e
```

Namely, ifFun takes three arguments, the first one being a boolean value. If the boolean is **True**, then ifFun returns its second argument; if the boolean is **False**, then ifFun returns its third argument. Hence, the second and the third arguments represent the two expressions following the **then** and **else** keywords in a conditional expression. Let us test `ifFun` against primitive conditional expressions:

```
> ifFun True 1 2 == if True then 1 else 2
True
> ifFun False 1 2 == if False then 1 else 2
True
```

Observe from the type of `ifFun` that the second and third arguments must have the same type, since either one or the other is returned. But this is exactly the same kind of type constraint over conditional expressions, where the sub-expressions in the two branches must have the same type. ■

7.3 Type inference and most general type

Consider the function `first` we introduced in Example 7.1. There we have declared `first` to have type `a → b → a`, but this is not the only valid type for `first`. Apart from the obvious renaming of type variables, which does not practically affect the type, we can declare `first` also with type:

```
first' :: a → a → a
first' x _ = x
```

or even with type:

```
first'' :: Int → a → Int
first'' x _ = x
```

or, if we like, with type:

```
first''' :: Int → Bool → Int
first''' x _ = x
```

and so on. All of these definitions of `first` are correct and well typed, they only differ in the constraints they impose on the type of the arguments. The first version of `first` does not impose any constraint, and this is testified by the fact that the two arguments are typed by two distinct type variables "`a`" and "`b`". The second version, `first'`, is still polymorphic, but the type is `a → a → a`, meaning that `first'` can only be applied to arguments *with the same type*. The third version, `first''` is somewhat more specific, in that the first argument must have type **`Int`**, whereas there is no constraint on the type of the second argument. In the last version, `first'''`, the function is no longer polymorphic, the first argument must have type **`Int`** and the second argument must have type **`Bool`**.

All these apparently unrelated, alternative types for `first` are related with the original type `a → b → a` by means of a strong property: they are all instances of the original type. For example, `a → a → a` is obtained from `a → b → a` by instantiating the type variable "`b`" with the type variable "`a`"; the type `Int → a → Int` is obtained from `a → b → a` by simultaneously instantiating the type variables "`a`" and "`b`" in the original type with the types **`Int`** and "`a`"; finally, the type `Int → Bool → Int` is ontained by instantiating "`a`" with **`Int`** and "`b`" with **`Bool`**. In fact, it is possible to show that `a → b → a` is the *most general* (or *principal*) type of `first`. Any other valid type of `first` is obtained by a suitable instantiation of the principal type.

We have said many times that Haskell is able to infer automatically the type of names defined in a script. Now that we know that the same name can be given different types, this explanation is no longer satisfactory becase it does not say *which* type, among the possibly infinite ones that are valid, is chosen by Haskell. The answer is that Haskell infers the principal type. As an experiment, we can try to define a typeless version of `first`, thus:

```
first x _ = x
```

and then ask `GHCi` to tell us its type:

```
> :type first
first :: t → t1 → t
```

`GHCi` has inferred the principal type of `first`, which is the type we have given to `first` in the first place. If desirable, the programmer can always provide an explicit type declaration for restricting the type of defined names.

7.4 Rigid type variables

When defining a function such as

```
f :: a → a
f x = E
```

the function `f` has a polymorphic type, meaning that it can be applied in different contexts with arguments of different type. When looking at the definition of the function `f`, it is legitimate to think that, in the expression E, the identifier `x` has type `a`. However, this does not mean that `x` has a polymorphic type. In particular, the type variable `a` occurring in the type of `f` is *rigid* within the body of `f` and if we try to use `x` in different contexts with different types, `GHCi` will complain. Loading the code

```
f :: a → a
f x = if x then x else x
```

into `GHCi`, where one particular occurrence of `x` is the test of a conditional expression, generates the error

```
   Couldn't match expected type ‘Bool’ against inferred type ‘a’
     ‘a’ is a rigid type variable bound by
         the type signature for ‘f’ at rigidness.hs:1:5
   In the predicate expression: x
   In the expression: if x then x else x
   In the definition of ‘f’: f x = if x then x else x
```

If this code were allowed, we could easily break `GHCi` by asking it to compute

```
  f 1 ⟹ if 1 then 1 else 1
```

which results in a ill-typed expression that cannot be reduced further. By declaring that the type of `x` contains a rigid type variable, we require that *all* the occurrences of `x` in the body of `f` must have the same type. In particular, the first occurrence of `x` cannot have type **`Bool`**, as required by the conditional expression.

A subtle consequence of rigid type variables is that there are certain functions that do not use their arguments and yet cannot be given a polymorphic type. Consider the following function definition:

```
f :: a → a
f x = if g True then g x else x
    where
    g y | False = y
        | otherwise = x
```

The function `f` declares a local function `g` which essentially is the constant function that always returns `x` (the argument of `f`) since the guard in the first equation of `g` is **`False`**. The function `g` does not use its argument in any way, except possibly returning it (even though this can never happen in practice). However, having both `y` and `x` on the right hand sides of the two equations for `g` has an important side-effect: the type of `y` and the type of `x` must be the same (this is similar to what happens in Example 7.2, where the two arguments of `ifFun` must have the same type since either one or the other can be returned). But the fact that type of `x` contains a rigid type variable somehow “contaminates” the type of `g` as well. Thus, `g` has type `a → a` but this type is not truly polymorphic, because `a` is a rigid type variable. As a consequence, the two occurrences of `g` in the body of `f` must have the same type, and this generates the same error we have seen before for rigid type variables.

7.5 Overloading and restricted polymorphism

The careful reader will have certainly noticed the strong similarity between the types of polymorphic and overloaded functions. Compare:

```
> :type id
id :: a → a
> :type (+)
(+) :: (Num a) ⇒ a → a → a
```

In fact, the type of the overloaded function `(+)` looks very much alike that of a polymorphic function, except for the context annotation **`Num`** `a`. What it says is that `(+)` accepts two arguments whose type is an almost arbitrary “a”, provided that the type `a` is an instance of the **`Num`** class.

As it happens, in Haskell polymorphism and overloading are somewhat related. In both cases type variables are used for denoting arbitrary types. However, in polymorphic functions type variables stand for *any type*, whereas in overloaded functions type variables stand for *any type that is an instance of a given class*. In this sense Haskell overloading can be thought of as a restricted form of polymorphism.

7.6 Exercises

Exercise 7.1. Infer the most general type of the following functions and check your answers with `GHCi`:

1. `f x y z = z`
2. `f x y z = if x then y else z`
3. `f x y z = if x == y then z else x`
4. `f x y u v = x == y || u == v`

Exercise 7.2. Find the most general type in each of the following sets of types:

1. `a → a → b`, `a → b → b`, `a → b → c`

2. **Int** → **Int** → **Bool**, a → b → c, **Int** → a → b
3. a → a, a → b, **Int** → **Int**
4. (**Integral** a) ⇒ a → a → a, **Int** → **Int** → **Int**, (**Eq** a) ⇒ a → a → a
5. **Int**, (**Integral** a) ⇒ a, (**Ord** a) ⇒ a

Exercise 7.3. For each type in Exercise 7.2 define a Haskell function having that as most general type.

Part III

Predefined structured data types

Chapter 8

Pairs and tuples

So far we have dealt with numbers, boolean values, and functions for transforming them. It is common in programming to group data in compound data structures. For example, it may be more efficient to have a function that returns *two* values rather than having two functions that return the two values separately. Or, it may be the case that a collection of basic values concretely represents a single value in an abstract data type. For instance, we may decide to represent rational numbers $\frac{m}{n}$ as pairs of integer numbers (m, n). This chapter covers pairs and tuples in Haskell.

8.1 Creating pairs and tuples

In Haskell, it is possible to create *pairs*, and more generally *tuples*, by writing a sequence of expressions separated by commas and enclosed within parentheses. For example

```
> (1, 2)
(1,2)
```

is a pair made of two components, the numbers 1 and 2, in this order, whereas

```
> (True, False, False)
(True,False,False)
```

is a triple made of the boolean values **True**, **False**, and **False**, in this order.

The value (**True**, **False**, **False**) is something that we have not encountered before, it is a value of a new type:

```
> :type (True, False, False)
(True, False, False) :: (Bool, Bool, Bool)
```

The type (**Bool**, **Bool**, **Bool**) denotes that (**True**,**False**,**False**) is a triple, where each component of the triple has type **Bool**. In general, the type

$(T_1, T_2, \ldots, T_n)$

where $n > 1$ denotes the type of a n-tuple (a tuple with n components) where the i-th component is of type T_i. Different components need not have same type. It is perfectly legal to create a triple with components of different types:

```
> (1, 2.0)
(1,2.0)
> (1, True, 2.0)
(1,True,2.0)
```

and since functions are values, it is also possible to create tuples with functions inside them:

```
> :type (True, not)
(True, not) :: (Bool, Bool → Bool)
```

although GHCi is unable to properly show them, since function values cannot be shown.

Tuples can be nested within tuples. We can have, for example

```
> (1, (True, False, False))
(1,(True,False,False))
> ((1, True), False, False)
((1,True),False,False)
> (1, (True, False), False)
(1,(True,False),False)
```

Observe that the three tuples above are not only different, but they also have different types:

```
> :type (1, (True, False, False))
(1, (True, False, False)) :: (Num t) ⇒ (t, (Bool, Bool, Bool))
> :type ((1, True), False, False)
((1, True), False, False) :: (Num t) ⇒ ((t, Bool), Bool, Bool)
> :type (1, (True, False), False)
(1, (True, False), False) :: (Num t) ⇒ (t, (Bool, Bool), Bool)
```

In particular, nested tuples are not “flattened”, their nesting structure is preserved.

Example 8.1. Suppose we need to sort 4 integer numbers in ascending order. The idea is to write a function sort4 that accepts 4 arguments and returns a quadruple containing the 4 arguments properly sorted. We can implement sort4 as follows:

```
sort4 :: Int → Int → Int → Int → (Int, Int, Int, Int)
sort4 a b c d | a < b && b < c && c < d = (a, b, c, d)
              | a > b = sort4 b a c d
              | b > c = sort4 a c b d
              | otherwise = sort4 a b d c
```

```
> sort4 4 2 3 1
(1,2,3,4)
```

The first equation in the definition of sort4 is guarded by the condition a < b && b < c && c < d which verifies whether the 4 arguments are sorted already. In this case, it suffices to create the quadruple with the 4 arguments. If the first argument is bigger than the second (second equation), then we can get closer to the right answer by “swapping” the first and the second arguments and recursively applying sort4. If the first argument is smaller than the second, but the second is bigger than the third (third equation), we swap the second and third arguments. In the remaining case (last equation), it must be that the third argument is bigger than the fourth one, while the others are correctly sorted already. Hence we do the swapping on the third and fourth arguments. ■

8.2 Deconstructing tuples with pattern matching

The Haskell standard library provides the utility functions **fst** and **snd** for respectively extracting the first and the second component of a pair. Thus we can write

```
> fst (1, 2)
1
> snd (1, 2)
2
```

The functions **fst** and **snd** are polymorphic, hence they can be applied to pairs of arbitrary type:

```
> :type fst
fst :: (a, b) → a
> :type snd
snd :: (a, b) → b
> fst (True, not)
True
> snd (1, 2.0)
2.0
```

The type of **fst** shows that **fst** can be applied to a pair of type `(a, b)` whose components have type `a` and `b`, and returns a value of type `a`, which is indeed the type of the first component. The type of **snd** is the symmetric of the type of **fst**. But the point is that **fst** and **snd** can only be applied to pairs, not to generic tuples:

```
> fst (True, False, False)
<interactive>:1:4:
    Couldn't match expected type ‘(a, b)’
           against inferred type ‘(Bool, Bool, Bool)’
    In the first argument of ‘fst’, namely ‘(True, False, False)’
    In the expression: fst (True, False, False)
    In the definition of ‘it’: it = fst (True, False, False)
```

What is needed is a more general mechanism for accessing the components of a tuple. This mechanism is a natural generalization of pattern matching (Section 5.3). Here is a function that accesses the two components of a pair of integer numbers and adds them together:

```
plusP :: (Int, Int) → Int
plusP (m, n) = m + n
```

The idea is that `(m, n)` on the left hand side of `=` is a *pattern*, namely a specification of the shape of the value that the function `plusP` has been applied to. Since the domain of `plusP` is `(Int, Int)`, it can only be applied to values whose shape is `(m, n)` for some `m` and `n`. Thus, the pattern specifies not only the shape of the value, but possibly the names of subparts of the value (in this case the two components of the tuple) that can be used on the right hand side of `=`.

```
> plusP (1, 2)
3
> plusP (2, 3)
5
```

Pattern matching generalizes without problems to tuples of arbitrary length:

```
fst3 :: (a, b, c) → a
fst3 (x, _, _) = x
```

Observe that, in this function definition, the second and third components of the triple are not used, hence they are matched by the pattern `_` without giving them a name.

Using pattern matching, the library functions **fst** and **snd** can be easily defined:

```
fst :: (a, b) → a
fst (x, _) = x

snd :: (a, b) → b
snd (_, x) = x
```

It is worth observing how `plusP` could be implemented using **`fst`** and **`snd`**:

```
plusP' : (Int, Int) → Int
plusP' p = let m = fst p
               n = snd p
           in m + n
```

Even though `plusP` and `plusP'` are equivalent, `plusP'` has a sort of "operational" look and is a typical implementation of unexperienced functional programmers: "extract the first component of the pair `p` and call it `m`, extract the second component of the pair `p` and call it `n`, add `m` and `n`". Conversely, `plusP` is more declarative and direct: "the two components of `p` are `m` and `n`, add them together".

Pattern matching can also be used for accessing "deep" components of a tuple. For example, we can define the functional composition of **`fst`** and **`snd`** thus:

```
fstSnd :: (a, (b, c)) → b
fstSnd (_, (x, _)) = x
```

and symmetrically the functional composition of **`snd`** and **`fst`** thus:

```
sndFst :: ((a, b), c) → b
sndFst ((_, x), _) = x
```

8.3 Comparing tuples

Equality and inequality operators work seamlessly with tuples. Two tuples are equal if the corresponding components are:

```
> (1, 2) == (1, 2)
True
> (1, 2) == (2, 3)
False
> (True, False, False) == (True, False, True)
False
> (True, False, False) /= (True, False, True)
True
```

Since (==) and (/=) can only compare values of the same type, and since tuples with different numbers of components have different types, it is not possible to compare tuples with different types:

```
> (1, 2) /= (True, False, False)
<interactive>:1:10:
    Couldn't match expected type ‘(t, t1)’
           against inferred type ‘(Bool, Bool, Bool)’
    In the second argument of ‘(/=)’, namely ‘(True, False, False)’
    In the expression: (1, 2) /= (True, False, False)
    In the definition of ‘it’: it = (1, 2) /= (True, False, False)
```

Similarly, it is not possible to compare tuples whose components are function values, since function values cannot be compared.

Tuples are ordered by default by means of lexicographic order. This means that a tuple is smaller than another one (of the same type) if the leftmost component of the first tuple in which the two tuples differ is smaller than the corresponding component in the second tuple. So for instance we have:

```
> (True, False, False) < (True, True, False)
True
> (2, 1) < (1, 1000)
False
```

The other relational operators behave similarly.

8.4 Case study: rational arithmetic

Suppose we need to develop a library of rational numbers of the form $\frac{m}{n}$. As we hinted at the beginning of this chapter, we may decide to represent a rational number as a pair (m, n) of integer numbers where the first component of the pair is the *numerator* and the second component of the pair is the *denominator*. Observe that integer numbers are just rational numbers whose denominator is 1.

The first function we need creates a rational number from a numerator m and a denominator n. The reason why having this function is desirable, rather than writing down the pair (m, n), is that we should not allow a rational number whose denominator is 0. We define a function mkFrac to create a rational number as follows:

```
mkFrac :: Int → Int → (Int, Int)
mkFrac m n | n /= 0 = (m, n)
```

Observe that this function is *partial*, namely it is defined on a proper subset of values in its domain. Indeed, there is no equation defining the behavior of mkFrac if we apply it to two numbers where the second one is 0, and if that happens an exception is raised:

```
> mkFrac 1 2
(1,2)
> mkFrac 3 4
(3,4)
> mkFrac 1 0
*** Exception: frac.hs:2:0-27: Non-exhaustive patterns in function mkFrac
%%
```

Then we might like to define a couple of functions for adding and multiplying fractions, as follows:

```
addFrac :: (Int, Int) → (Int, Int) → (Int, Int)
addFrac (a, b) (c, d) = simplFrac (a * d + b * c, b * d)

mulFrac :: (Int, Int) → (Int, Int) → (Int, Int)
mulFrac (a, b) (c, d) = simplFrac (a * c, b * d)

simplFrac :: (Int, Int) → (Int, Int)
simplFrac (a, b) = (a ‘div‘ d, b ‘div‘ d)
    where d = gcd a b
```

```
> addFrac (1, 2) (3, 4)
(5,4)
> mulFrac (1, 2) (2, 3)
(1,3)
```

Notice that we simplify a fraction after every operation so that it is reduced as much as possible. To do so, we have defined an auxiliary function `simplFrac` that divides numerator and denominator by their greatest common divisor. Instead of `euclid` (see Section 6.2) we use the predefined library function **`gcd`**.

When comparing rational numbers, the default lexicographic order of tuples provided by Haskell may not work well. For example, we have

```
> mkFrac 1 2 < mkFrac 1 3
True
```

despite $\frac{1}{2} > \frac{1}{3}$. What we need is a specialized comparison function for pairs representing fractions:

```
ltFrac :: (Int, Int) → (Int, Int) → Bool
ltFrac (a, b) (c, d) = a * d < c * b
```

```
> mkFrac 1 2 `ltFrac` mkFrac 1 3
False
> mkFrac 1 3 `ltFrac` mkFrac 1 2
True
```

This problem with the relational operators highlights a limit of this representation of rational numbers. The fact that we rely on an existing data type for representing fractions prevents us from giving relational operators a specific meaning: we have to rely on their behavior when applied to pairs of numbers. If we need a specific meaning for some operator, such as <, we must provide a function such as `ltFrac`. In doing so, however, we loose part of the beauty of the notation, since we are forced to use `` `ltFrac` `` in place of <. We will see in Chapter 15 how to define a *new* type that is isomorphic to `(Int, Int)`. This allows the programmer to give < the desired meaning when used for comparing fractions.

8.5 Case study: faster Fibonacci

Consider the following 2×2 matrix:

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

It is a trivial exercise to prove (by induction on n) that

$$A^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

where we write F_n for the n-th Fibonacci number in the sequence shown in Example 6.2. This result gives us an alternative, more efficient way of computing the n-th Fibonacci number since we know, from Example 6.3, that A^n can be computed using just $\log_2 n$ multiplications. The only difference is that in Example 6.3 we were multiplying integer numbers, while here we are multiplying 2×2 matrices, but the asymptotic complexity of the computation is still logaritmic in n.

A 2×2 integer matrix can be represented as a tuple with 4 components of type **Int**. With this insight we can write the function that multiplies two matrices as follows:

```
mulMatrix :: (Int, Int, Int, Int) → (Int, Int, Int, Int)
                                  → (Int, Int, Int, Int)
mulMatrix (a11, a12, a21, a22) (b11, b12, b21, b22) =
    (a11 * b11 + a12 * b21, a11 * b12 + a12 * b22,
     a21 * b11 + a22 * b21, a21 * b12 + a22 * b22)
```

Now we can rephrase the power function in Example 6.3 so that it multiplies matrices as opposed to integer numbers, recalling that $A^0 = I$ where I is the identity matrix:

```
powMatrix :: (Int, Int, Int, Int) → Int → (Int, Int, Int, Int)
powMatrix _ 0 = (1, 0, 0, 1)
powMatrix a n | even n = a' `mulMatrix` a'
              | otherwise = a `mulMatrix` a' `mulMatrix` a'
    where
      a' = powMatrix a (n `div` 2)
```

The function that computes the n-th Fibonacci number can now be implemented by applying powMatrix to the tuple representing A and to the number n. Then, it suffices to extract the second component of the resulting tuple in order to obtain the n-th Fibonacci number:

```
fasterFibo :: Int → Int
fasterFibo n = let (_, x, _, _) = powMatrix (1, 1, 1, 0) n in x
```

Observe that the only way to extract this component is by means of pattern matching, but in this context we are not applying the tuple to any function. Rather, we have obtained the tuple from the application of a function and we merely need its second component. The expression

```
  let (_, x, _, _) = powMatrix (1, 1, 1, 0) n in x
```

is similar to a local definition (see Section 4.2) except that we have used a pattern rather than a single name to the left hand side of =. The value of the expression to the right hand side of = is evaluated, its value is matched against the pattern, and the resulting bindings can be used in the expression following the keyword **in**.

A few tests suffices to convince ourselves that the implementation is correct:

```
> fasterFibo 0
0
> fasterFibo 7
13
> fasterFibo 11
89
```

8.6 Exercises

Exercise 8.1. Determine the most general type of the following well-typed expressions:

1. `(1, 2, 3.5)`
2. `((==), 1)`
3. `((==) 1, 1)`

4. `((==) (1, 1))`
5. `((1, 1.5), (2, 1.5))`
6. `((+), 1)`
7. `((+) 1, 1)`
8. `(id, (+) 1, (/) 2)`
9. `(($), id)`
10. `((fun x y → (x, y)), (fun x y → (y, x)))`

Exercise 8.2. Determine the simplest pattern, with the least number of variables, to bind the number 14 in the following values to the variable x:

1. `(14, 1)`
2. `((true, (1, 14)), false, 1.5)`
3. `((false, 1.5), (true, 14))`
4. `(false, 1.5, true, 14)`
5. `((false, 1.5, true), 14)`

Exercise 8.3. Define a function that, when applied to a triple (a, b, c) rotates it to the left by returning the triple (b, c, a).

Exercise 8.4. Define a function to sort the elements of a 4-tuple in ascending order (hint: instead of attempting to produce the sorted tuple in one go, define a recursive function that progressively sorts the tuple).

Exercise 8.5. Using the representation of rational numbers $\frac{m}{n}$ as pairs of integer numbers (m, n), define conversion functions from rational numbers to floating-point numbers and viceversa. For the latter conversion, apply an arbitrary approximation.

Exercise 8.6. Represent complex numbers $a + bi$ as pairs of floating-point numbers (a, b). Define functions for adding, subtracting, multiplying, and dividing complex numbers using this representation.

Chapter 9

Lists

The number of components of a tuple must be statically known at the time a program is compiled, since this number is determined by the very type of the tuple. This makes tuples unsuitable for collecting values when the number of values in not known until the program is run.

9.1 Creating lists

A *list* is a uniform, ordered aggregation of values. The aggregation is uniform in the sense that all the values belonging to the same list must have the same type. The aggregation is ordered in the sense that the order in which values are aggregated is relevant (thus lists are *not* like sets in mathematics).

The easiest way of creating a list is by enumerating its components separated by commas and enclosed within square brackets. For example

```
> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0,1,2,3,4,5,6,7,8,9]
```

is the list containing the first 10 natural numbers. Whereas

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

is a list containing the values **True**, **False**, and **False**, in this order.

The *empty list* is the list containing no value:

```
> []
[]
```

The type of lists containing values of type T is written `[`T`]`. Unlike tuples, the type of a list says nothing about the number of elements in the list. For example, we have:

```
> :type [0, 1, 2]
[0, 1, 2] :: (Num t) ⇒ [t]
> :type [0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5] :: (Num t) ⇒ [t]
> :type [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] :: (Num t) ⇒ [t]
> :type [True, False, False]
[True, False, False] :: [Bool]
> :type []
[] :: [a]
```

We see that the first three lists have an overloaded type, since as we know the integer literals are overloaded. Also, the first three lists have the same type, despite they contain different numbers of elements. The empty list has a polymorphic type: since it contains no element, GHCi is unable to give a more precise type for it.

Lists can contain values of any type, provided that each element has the same type. In particular, it is legal to have lists within lists, tuples within lists, and functions within lists:

```
> :type [[], [True, False, False], [False]]
[[], [True, False, False], [False]] :: [[Bool]]
> :type [(True, 1), (False, 1), (False, 2)]
[(True, 1), (False, 1), (False, 2)] :: (Num t) ⇒ [(Bool, t)]
> :type [(&&), (||)]
[(&&), (||)] :: [Bool → Bool → Bool]
```

Given two lists of the same type, it is possible to *concatenate* them by means of the infix operator ++. Concatenation produces a *new* list containing all the elements of the first list followed by all the elements of the second list. For example:

```
> [1, 2, 3] ++ [4, 5, 6]
[1,2,3,4,5,6]
> [True] ++ [False, False]
[True,False,False]
> [] ++ [True, False, False]
[True,False,False]
```

As shown by the last example, the empty list [] is the neutral element for ++.

Like for any other binary operator, (++) stands for a function with two arguments for concatenating lists. Its type

```
> :type (++)
(++) :: [a] → [a] → [a]
```

clearly shows that the two lists being concatenated must have the same type.

Example 9.1. Let us define a function for creating lists of integer numbers within a specified range. The function takes two arguments m and n representing the lower and the upper limit of the range:

```
fromTo :: Int → Int → [Int]
fromTo m n | m > n = []
           | otherwise = [m] ++ fromTo (m + 1) n
```

The base case is when the lower limit m is greater than the upper limit n. Since there is no number in this range, the empty list is returned. When the lower limit is smaller than (or equal to) the upper limit, the list begins with the lower limit [m] followed by all the integer numbers from m + 1 to n:

```
> fromTo 0 9
[0,1,2,3,4,5,6,7,8,9]
> fromTo (-5) 5
[-5,-4,-3,-2,-1,0,1,2,3,4,5]
```

In the last expression the parentheses in (-5) are necessary for avoiding the known issues about operator precedence and function application. ■

9.2 Canonical list constructors

Observe that there can be *different* ways of creating the *same* list using ++. For example, the list [1,2,3] can be obtained in one of the following ways

```
> [] ++ [1, 2, 3]
[1,2,3]
> [1] ++ [2, 3]
[1,2,3]
> [1, 2] ++ [3]
[1,2,3]
> [1, 2, 3] ++ []
[1,2,3]
```

not counting the fact that the empty list [] can be concatenated (to the left or to the right) an arbitrary number of times without affecting the result. This makes ++ a *non-canonical constructor* of lists, which is unfortunate because it means that we cannot use ++ to understand how a list has been built, which is essential as soon as we want to perform pattern matching over lists.

For this reason, Haskell also provides a *canonical constructor* for nonempty lists denoted by the symbol : and usually referred to as "cons". The cons operator takes an element v and a list l and creates a new list $v : l$ beginning with v followed by all the elements in l. The element v is called the *head* of the list, whereas the old list l is called the *tail* of the list. For example we have

```
> 1 : []
[1]
> True : []
[True]
```

We can create all of the lists shown earlier using repeated occurrences of : and one final []. For example:

```
> 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : []
[0,1,2,3,4,5,6,7,8,9]
> True : False : False : []
[True,False,False]
```

Since : is right-associative, there is no need to write parentheses and 1 : (2 : []) can be safely written as 1 : 2 : [].

Remember that : takes an element on its right and a list of elements on its left. This is shown clearly the type of (:):

```
> :type (:)
(:) :: a → [a] → [a]
```

We see that (:) is a polymorphic function accepting an argument of type 'a' – the head of the list being created – and another argument of type [a] – the tail of the list being created – and returning a list of type [a].

Example 9.2. The function fromTo in Example 9.1 can be rewritten using the canonical constructor : instead of the concatenation operator ++:

```
fromTo :: Int → Int → [Int]
fromTo m n | m > n = []
           | otherwise = m : fromTo (m + 1) n
```

■

9.3 Deconstructing lists with pattern matching

Every list is either the empty list [] or has the form $v : l$ for some head v and some tail l. For this reason [] and : are the canonical list constructors that can be used in patterns for defining functions that analyze and deconstruct lists. For example, the function

```
headOrZero :: [Int] → Int
headOrZero [] = 0
headOrZero (x : _) = x
```

returns the head of a list of integers if the list is not empty and returns 0 otherwise. The first equation matches the list against the pattern []: there is only one list that matches this pattern, namely the empty list. The second equation matches the list against the pattern (x : _). Every non-empty list matches this pattern, since a non-empty list has a head (here captured by the pattern variable x) and a tail (which is matched here by _). So we have

```
> headOrZero []
0
> headOrZero [1, 2, 3]
1
```

Observe that in the definition of headOrZero we could have changed the order of the two equations without affecting the behavior of the function. This happens because the two patterns do not *overlap*: no non-empty list matches [], and the empty list does not match (x : _).

The head and tail of a list can be "extracted" by means of two simple functions:

```
head :: [a] → a
head (x : _) = x

tail :: [a] → [a]
tail (_ : xs) = xs
```

Observe that both **head** and **tail** are partial functions: they can only be applied to non-empty lists, since the empty list does not have a head nor a tail. Thus

```
> head [True, False, False]
True
> tail [True, False, False]
[False,False]
```

but

```
> head []
*** Exception: Prelude.head: empty list
```

Example 9.3 (length of a list). As we have said the length of a list is not encoded in its type, so it would be useful to have a function to *compute* it. The idea is to scan a list (or better its canonical representation in terms of : and []) until we find []. This is a typical case in which a recursive function is needed. The only difference with respect to other recursive functions defined over numbers is that the base case is represented by the empty list, and the recursive case is represented by the non-empty list:

```
length :: [a] → Int
length [] = 0
length (_ : xs) = 1 + length xs
```

The length of the empty list is obviously `0`. The length of a non-empty list is one plus the length of its tail:

```
> length [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
10
> length [True, False, False]
3
> length (fromTo (-5) 5)
11
```

Observe that `lenght` is a polymorphic function: it does not depend in any way on the type of the elements of the list it is applied to. ■

Example 9.4 (list concatenation). It is easy to implement ++ in terms of `:`, as follows:

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

The recursion merely involves the first of the two arguments of `append`. For example:

```
  append (1 : 2 : 3 : []) (4 : 5 : 6 : []) ⟹
  1 : append (2 : 3 : []) (4 : 5 : 6 : []) ⟹
  1 : 2 : append (3 : []) (4 : 5 : 6 : []) ⟹
  1 : 2 : 3 : append [] (4 : 5 : 6 : []) ⟹
  1 : 2 : 3 : 4 : 5 : 6 : []
```

Observe once again that the type of `append` enforces the two arguments to be lists of the same type. ■

Having two constructors (++ and `:`) for lists may appear confusing, especially since one can be defined in terms of the other. In practice, it is more useful to use one or the other depending on the context. The following example shows a function that can be more conveniently (although inefficiently) implemented using ++ rather than `:`.

Example 9.5 (reverse of a list). Suppose we are to define a function `rev` that, as the name suggests, reverses the order of the elements in a list. Namely, when applied to a list `rev` returns another list with the same elements of the original list, but in the inverse order.

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

The function `rev` must be recursive: the base case is when the list is empty, since the reverse of the empty list is the empty list itself. When the list is not empty, namely it has the form `(x : xs)` for some head `x` and some tail `xs`, the idea is to append `x` at the very end of the reverse of `xs`. Since `:` is used to *prepend* an element at the beginning of a list, it cannot be used in this example.

```
> rev (fromTo (-5) 5)
[5,4,3,2,1,0,-1,-2,-3,-4,-5]
```

■

9.4 Comparing lists

Just as with tuples, Haskell equips lists with default relational operators. Equality and inequality work as expected: if two lists have different lengths, then they are necessarily different; if they have the same length, then they are equal provided that corresponding elements are pairwise equal:

```
> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] == fromTo 0 9
True
> fromTo 0 9 == []
False
> [True, False, False] /= []
True
```

Unlike tuples, relational operators can compare lists of different lengths, provided that they have the same type. The lexicographic ordering of lists states that a list l is smaller if a list l' if either l is a prefix of l' or if the first element of l in which l and l' disagree is smaller than the corresponding element in l'. Thus we have

```
> fromTo 0 9 < fromTo 0 100
True
> fromTo 0 9 < []
False
> [1, 2, 3, 4, 5] < [4, 5]
True
```

In the last example, observe that the list `[1, 2, 3, 4, 5]` is smaller than the list `[4, 5]` according to <, despite being a longer list. This is because the first element of the longer list, `1`, is smaller than the first element of the shorter list, `4`, hence the longer list is smaller than the longer one according to the lexicographic order.

9.5 List transformations

Very often it is necessary to transform one or more lists into another list. The most frequent kinds of transformations are uniform element-wise transformations (trasforming every single element of a list in a uniform way) or element filtering (removing every element that does not satisfy certain properties). For example, we can write a recursive function that computes the successor of every element in a list of integer thus:

```
mapSucc :: [Int] → [Int]
mapSucc [] = []
mapSucc (x : xs) = x + 1 : mapSucc xs
```

and similarly we can remove every odd element from a list of integers thus:[1]

```
filterEven :: [Int] → [Int]
filterEven [] = []
filterEven (x : xs) | x `mod` 2 == 0 = x : filterEven xs
                    | otherwise = filterEven xs
```

Haskell provides a handy notation for expressing (combinations of) these transformations. For example, `mapSucc` and `filterEven` can be expressed using *list comprehensions* as

```
mapSucc :: [Int] → [Int]
mapSucc xs = [ x + 1 | x ← xs ]
```

[1]When we write “remove every odd element from a list l” we actually mean “creating another list with only the even elements of l”.

```
filterEven :: [Int] → [Int]
filterEven xs = [ x | x ← xs, x ‘mod‘ 2 == 0 ]
```

In general the list comprehension

```
[ f x | x ← l ]
```

stands for a list obtained by iterating over all the elements x of the list *l*, and applying the function *f* to x. The fragment x ← *l* of a list comprehension is called *generator*. For example:

```
> [ x + 1 | x ← fromTo 0 9 ]
[1,2,3,4,5,6,7,8,9,10]
> mapSucc (fromTo 0 9)
[1,2,3,4,5,6,7,8,9,10]
```

The list comprehension

```
[ x | x ← l, p x ]
```

stands for the sublist of *l* obtained by iterating over all the elements x of *l* such that *p* x == **True**. The part *p* x is called a *filter*. For example:

```
> [ x | x ← fromTo 0 9, x ‘mod‘ 2 == 0 ]
[0,2,4,6,8]
> filterEven (fromTo 0 9)
[0,2,4,6,8]
```

As another example, creating the list of the prime numbers smaller than 100 is as simple as:

```
> [ x | x ← fromTo 0 100, prime x ]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

and the function that computes the divisors of an integer number can be defined thus:

```
divisors :: Int → [Int]
divisors n = [ x | x ← fromTo 1 n, x ‘divides‘ n ]
```

List comprehensions are not limited to one generators. Several generators can be used for combining multiple lists into a single one. For instance

```
> [ (x, y) | x ← fromTo 0 2, y ← [False, True] ]
[(0,False),(0,True),(1,False),(1,True),(2,False),(2,True)]
```

creates all possible combinations of an integer number from 0 to 2 and a boolean value.

Generators may depend on each other. For example

```
> [ (x, y) | x ← fromTo 0 2, y ← fromTo 0 x ]
[(0,0),(1,0),(1,1),(2,0),(2,1),(2,2)]
```

creates all pairs of integer values between 0 and 2 such that, in each pair, the first component is bigger than or equal to the second component.

Example 9.6. In Chapter 6 we have computed the volume of a pyramid by summing up the volumes of all its levels. If we define a function `sumInt` to sum up all the elements in a list of integers, then we can provide an alternative solution to the same problem:

```
pyramid’ :: Int → Int
pyramid’ n = sumInt [ x * x | x ← fromTo 1 n ]
```

The advantage of this solution is in its modularity: `pyramid`, which solves a specific problem, is composed of smaller functions (such as `sumInt` and `fromTo`) that are generally useful and can be applied in different contexts.

Computing the sum of the elements of a list of integers is quite simple:

```
sumInt :: [Int] → Int
sumInt [] = 0
sumInt (x : xs) = x + sumInt xs
```

If the list is empty, then there is no element to add to the sum hence the result is `0`. If the list has a head `x` and a tail `xs`, then we compute the sum of all the elements in `xs` (which is shorter than the original list) and then we add `x`. Compare

```
> pyramid 100
338350
> pyramid’ 100
338350
```

■

Example 9.7. We can provide an alternative implementation of the function `fact` (Example 6.1) along the same lines of the previous example. In this case, we want to compute the product of a list of integer numbers:

```
prodInt :: [Int] → Int
prodInt [] = 1
prodInt (x : xs) = x * prodInt xs
```

The difference between `prodInt` and `sumInt` in the previous example is that we use `*` instead of `+` and that we let the product of zero numbers be `1`. The factorial is now as easy as

```
fact’ :: Int → Int
fact’ n = prodInt (fromTo 1 n)
```

Observe that this version works correctly even in the case when `n` is `0`. ■

Example 9.8 (combining lists). List comprehensions cannot be used for transforming corresponding elements of two or more lists, since it is not possible to easily express the fact that two elements of two different lists are in the same position by means of a filter. In these cases, it is useful to *combine* two (or more) lists into a single list of pairs (or tuples), and the to process the resulting list. The **`zip`** function, which is predefined in the standard Haskell library, does just this:

```
zip :: [a] → [b] → [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

Observe that **`zip`** is a total function, and combines the corresponding elements of two lists as much as possible. If one list is longer than the other, the residual elements are ignored. ■

9.6 Case study: association lists

Maps and dictionaries are ubiquitous in computing. A map or dictionary is a data structure that associates keys from a (typically finite) domain with values. For example, the symbol table of a compiler, that associates names of identifiers occurring in a program with the information related to them (such as their type, their location in memory, their allocation strategy, and so on), is a perfect example of dictionary. Maps and dictionaries always come along with a set of operations to access and modify them. The most common operations include: looking for the value associated with a key (**lookup**); checking whether there is an association for a key (member); inserting and deleting associations (**insert** and **delete**).

In Haskell, maps can be easily (although not terribly efficiently) implemented using a combination of lists and pairs. In particular, a list with type `[(a, b)]` is a list of pairs whose first component has type 'a' and whose second component has type 'b'. Thus, a value

$[(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)]$

represents a finite map associating they key k_i with the corresponding value v_i. For example, the script

```
l1 :: [(String, Int)]
l1 =
  [("monday", 1), ("tuesday", 2), ("wednesday", 3),
   ("thursday", 4), ("friday", 5), ("saturday", 6), ("sunday", 7)]
l2 :: [(Int, String)]
l2 =
  [(1, "monday"), (2, "tuesday"), (3, "wednesday"),
   (4, "thursday"), (5, "friday"), (6, "saturday"), (7, "sunday")]
```

defines two maps associating week days with numbers and viceversa (we will talk more about strings in the next chapter).

Then, it is easy to write a Haskell function that retrieves the value associated with a given key into a map:

```
lookup :: (Eq a) ⇒ a → [(a, b)] → a
lookup k ((k', x) : _) | k == k' = x
lookup k (_ : ps) = assoc k ps
```

Namely **lookup** k l scans the list l until a pair whose first component is equal to k is found. At that point, the second component of the pair is returned. If no such pair exists, the function terminates abnormally by raising an exception. Observe that **lookup** is polymorphic in both the type of the keys and the type of the values. The only requirement is, obviously, for keys to belong the **Eq** type class since we are using the == operator over them. Then, **lookup** could be equally applied to l1 or l2, since both these values are lists of pairs whose first component has a type that belongs to the class **Eq**.

The insertion of a new association into an existing map simply prepends the relevant pair in front of the association list. Since the **lookup** function we have defined above scans the list starting from its head, this insertion has indeed the effect of overriding any existing association for the same key:

```
insert :: a → b → [(a, b)] → [(a, b)]
insert k x ps = (k, x) : ps
```

Similarly, one could write functions to remove associations or to test the presence of associations:

```
delete :: (Eq a) ⇒ a → [(a, b)] → [(a, b)]
delete _ [] = []
delete k ((k', _) : ps) | k == k' = ps
delete k (p : ps) = p : delete k ps

member :: (Eq a) ⇒ a → [(a, b)] → Bool
member _ [] = False
member k ((k', _) : _) = True
member k (_ : ps) = member k ps
```

9.7 Exercises

Exercise 9.1. Example 9.4 shows that ++ can be defined in terms of :. Define a function `cons` that behaves as : but is defined in terms of ++.

Exercise 9.2. Determine which of the following expressions are well typed and, for those that are well typed, determine the most general type:

1. `[1, 2, 3.5]`
2. `[1, 2, round 3.5]`
3. `[(==) 1, id]`
4. `[(==) 1, (==) 2.5]`
5. `[(1, 1.5), (2, 1.5)]`
6. `[( + ), ( * )]`
7. `[id, (+) 1; (/) 2]`
8. `[fst, (+)]`
9. `[($), id]`

Exercise 9.3. Define a function that takes an integer number n and a value v and creates a list containing n occurrences of v.

Exercise 9.4. Define a function that takes an integer number n and returns the list of the first n prime numbers.

Exercise 9.5. Define a function that, when applied to a triple, produces a list with all the permutations of the element in the triple.

Exercise 9.6 (**). What is the advantage in defining ++ as a right-associative operator? Hint: suppose that ++ is implemented as `append` in Example 9.4.

Exercise 9.7. Define a predicate to verifies whether a list is sorted in ascending order.

Exercise 9.8. Define a function that removes duplicated elements from a list and preserves the first occurrence of each element.

Exercise 9.9. Solve Exercise 9.8 more efficiently by assuming that the input list is sorted in ascending order.

Exercise 9.10. Define a function that, when applied to an integer number n and to a list l, returns the prefix of l of length at most n.

Exercise 9.11. Define a function **`insert`** that, when applied to a value v and a sorted list l, produces another sorted list obtained by adding v to l in the right position.

Exercise 9.12. Define a sorting algorithm based on the function **`insert`** in Exercise 9.11.

Exercise 9.13. A list l_1 is a sublist of another list l_2 if all the elements in l_1 (considering also the duplicate elements) occur in in l_2 in the same order, possibly interspersed with other elements of l_2. Define a function `sublist` that, when applied to l_1 and l_2, checks whether l_1 is a sublist of l_2.

Exercise 9.14. Define a function `merge` that, when applied to two lists sorted in ascending order, returns the sorted list obtained by merging the two original lists.

Exercise 9.15 (*). Define a function `half` that splits a list of length n into two sublists of length at most $\lceil n/2 \rceil$.

Exercise 9.16. Solve Exercise 9.15 making sure to scan the original list only once. Hint: use pairs.

Exercise 9.17. Use the function `merge` in Exercise 9.14 and the function `half` in Exercise 9.15 to implement the *merge sort* algorithm.

Exercise 9.18. Define a function `rotate` that, when applied to a list `[`a_1`, `a_2`, ..., `a_n`]`, returns the list `[`a_2`, ..., `a_n`, `a_1`]`.

Exercise 9.19. Look at the type and documentation of function **`zip`**. Provide an implementation of **`zip`** without looking at the source code of the standard library.

Exercise 9.20. Define a variant of the function **`delete`** in Section 9.6 that removes *all* the pairs in an association list with a given key, instead of the first one.

Chapter 10

Characters and strings

Many programs print messages on the terminal or in a window and read text from an input device or from a file. Before it is sent to a terminal or stored in a file, data is often converted to *strings of characters* that provide a textual representation for the data being written or read. In this chapter we introduce the **Char** type, we take advantage of the fact that Haskell strings are simply lists of characters for making some more practice with lists, and we see two more cases of function overloading.

10.1 Characters

Haskell provides a specific type **Char** for representing characters. As customary in many programming languages, character literals are denoted by a printable character enclosed within single quotes, thus:

```
> 'a'
'a'
> 'A'
'A'
```

Alternatively, it is possible to specify a character by means of its Unicode code expressed as decimal number or, if preceeded by a lowcase `x`, as an hexadecimal number:

```
> '\97'
'a'
> '\65'
'A'
> '\x41'
'A'
```

There are also common escape sequences for representing "invisible" characters, such as `'\n'` for "new line", `'\t'` for "tabulation", and so on. A full list can be found in the Haskell reference manual.

All character literals have type **Char**, which is incompatible with any other Haskell type:

```
> :type 'a'
'a' :: Char
> :type '\x41'
'\x41' :: Char
> 'a' == 97
<interactive>:1:7:
    No instance for (Num Char)
      arising from the literal ‘97’ at <interactive>:1:7-8
    Possible fix: add an instance declaration for (Num Char)
    In the second argument of ‘(==)’, namely ‘97’
    In the expression: 'a' == 97
    In the definition of ‘it’: it = 'a' == 97
```

The last expression is ill typed since we are trying to compare a character and a number literal. Often it is necessary or handy to manipulate characters as if they were numbers, for example by computing the character corresponding to a number. The standard Haskell library provides several functions for dealing with characters, including conversion functions from **Char** to **Int** and viceversa. All these functions are contained in the Data.**Char** module. For this reason, when using these functions in GHCi, it is necessary to prepend their name with the Data.**Char** prefix. For example

```
> Data.Char.toLower 'A'
'a'
```

converts the character A to lower case.

In standalone Haskell scripts, it is necessary to *import* the Data.**Char** module at the beginning of the script:

```
import Data.Char

a :: Char
a = toLower 'A'
```

We will say more about modules in Chapter 17. Going back to our conversion problem, we can use the functions **chr** and **ord** for respecively convering an integer value to the corresponding character, and a character to its Unicode as an integer number:

```
> Data.Char.chr 97
'a'
> Data.Char.ord 'a'
97
> 'a' == Data.Char.chr 97
True
```

10.2 Strings as lists of characters

In Haskell, strings are nothing but lists of characters. Thus, the type **String** is just an alias for the type [**Char**] meaning that the two types can be used interchangeably. The syntax of string literals is rather common: string literals are finite sequences of characters enclosed within double quotes. For example:

```
> "Hello,␣World!"
"Hello,␣World!"
> "H\97skell"
"Haskell"
```

GHCi does not distinguish between **String** and [**Char**] when printing the type of strings, hence we have:

```
> :type "Hello,␣World!"
"Hello,␣World!" :: [Char]
```

```
> :type "H\97skell"
"H\97skell" :: [Char]
```

Observe that, within strings, it is possible to use the same escape sequences that work for character literals.

Since strings are just lists of characters, functions and operators over lists can be used for strings as well, provided that the usual type constraints are satisfied. For example, it is possible to use `length` for computing the length of a string, and the concatenation operator ++ is also the concatenation operator for strings:

```
> length "Hello,␣World!"
13
> "Hello," ++ "␣World!"
"Hello,␣World!"
```

In addition, list comprehension can be used for performing string transformations as for lists. For example, we can remove spaces from a string thus:

```
> [ c | c ← "Hello,␣World!", not (Data.Char.isSpace c) ]
"Hello,World!"
```

and we can turn a whole string to upper case thus:

```
> [ Data.Char.toUpper c | c ← "Hello,␣World!" ]
"HELLO,␣WORLD!"
```

10.3 Case study: collapsing spaces

Suppose we want to define a function `collapse` that collapses all the sequences of consecutive spaces in a string. For example, `collapse` should behave like this:

```
> collapse "␣␣␣Hello,␣␣␣World!␣␣␣"
"␣Hello,␣World!␣"
```

In order to understand how to implement `collapse`, we should try to imagine an hypothetical automaton that scans the string from left to right, one character at a time. The automaton can sense whether the current character is a space or not (to this aim, we will use the `isSpace` function from the `Data.Char` module), it can decide to keep the character, or it can decide to drop the character. The automaton needs two different *states*, which we will call “find” and “drop”. When it is in the “find” state, the automaton has yet to find a space in the string, and keeps all the characters it sees. When the automaton is in the “drop” state, the automaton has found a space and hence drops any further space. We simulate the automaton with two functions, one for each state, and we implement `collapse` as follows:

```
import Data.Char

collapse :: String → String
collapse s = find s
    where
      find "" = ""
      find (c : s) | isSpace c = c : drop s
                   | otherwise = c : find s

      drop "" = ""
      drop (c : s) | isSpace c = drop s
                   | otherwise = c : find s
```

Both **find** and **drop** return the empty string if they reach the end of the original string. When **find** is applied to a non-empty string that begins with a space, that space is preserved and we process the rest of the string by means of **drop**. As long as **find** is applied to strings beginning with non-space characters, the characters are preserved and the automaton stays in the "find" mode. The **drop** function discards every space it sees (recall that by the definition of **find** the first space is preserved) and switches back to the "find" mode as soon as a non-space character is found.

Having **find** and **drop** in place, the function `collapse` needs just to start up the automaton in state "find". Observe that **find** and **drop** are an example of *mutually recursive* functions: **find** may apply **drop** and **drop** may apply **find**. This is perfectly legal as long as the recursion is well-founded.

10.4 Showing values

When building stand-alone Haskell programs that are not run within `GHCi` it is often necessary to print computed values on the terminal (or writing them in a text file). This requires the ability to convert Haskell values to strings of characters. Observe that (a) each type requires a specific procedure for converting values of that type into strings and (b) there are types (such as arrow types) for which there is no obvious conversion. This situation recalls very closely the problem of equality and, indeed, makes a perfect case for overloading. Haskell provides an overloaded function **show** for converting values into strings:

```
> :type show
show :: (Show a) ⇒ a → String
```

Thus **show** accepts values of type 'a' and returns strings, but the type 'a' must belong to the class **Show** which characterizes those types whose values admit conversion into strings of characters. All the types we have seen so far except arrow types belong to the **Show** class:

```
> show 1
"1"
> show 2.5
"2.5"
> show True
"True"
> show 'a'
"'a'"
> show "Hello,␣World!"
"\"Hello,␣World!\""
```

Also tuples and lists are "showable", provided that the contained values are showable as well.

Example 10.1. As a useful exercise, we may try to implement **show** for lists, assuming that the type of the elements of the list can be shown. To avoid confusion, we will call the function `myShowList` instead of **show**:

```
myShowList :: (Show a) ⇒ [a] → String
myShowList xs = "[" ++ aux xs ++ "]"
    where
      aux [] = ""
      aux [x] = show x
      aux (x : xs) = show x ++ "," ++ aux xs
```

All the interesting work is done in the auxiliary function `aux`, whose result is concatenated with a pair of balanced square brackets by `myShowList`. When `aux` is applied to the empty list, it returns the empty string since there is no element to show. When the list is not empty, `aux` should return the head of the list converted to a string, followed by the separator `","` followed by the tail of the list converted to a string. However, we must not show the separator if the tail of the list is empty. This is why `aux` has an intermediate case when the list is made of just one element: the element is converted to a string, but it is not followed by the separator.

We may compare `myShowList` and **show** with a few tests:

```
> myShowList [1, 2, 3] ++ "␣␣or␣␣" ++ show [1, 2, 3]
"[1,2,3]␣␣or␣␣[1,2,3]"
> myShowList [True, False] ++ "␣␣or␣␣" ++ show [True, False]
"[True,False]␣␣or␣␣[True,False]"
> myShowList [] ++ "␣␣or␣␣" ++ show []
"[]␣␣or␣␣[]"
```

■

10.5 Reading values

Haskell provides a function **read** that is somewhat the inverse of **show**: it interpretes a string as a literal constant and converts it into the corresponding Haskell value. The types whose values can be converted in this way must belong to the type class **Read**, as the type of **read** shows:

```
> :type read
read :: (Read a) ⇒ String → a
```

The **read** function is very different from **show**, and not simply because they implement inverse transformations. One additional difference is that **read** is necessarily *partial*: there are strings that cannot be interpreted as any value. But the most important difference regards **read**'s type: both **show** and **read** are overloaded functions. While **show**'s overloading depends on the type of the argument it is applied to – we have seen examples of this in Section 10.4 –, **read**'s overloading depends on the type of the value returned by **read**! In other words, the exact behavior of **read** does not depend on the type of its argument, which is invariably **String**, but rather on the type expected by the context in which the application of **read** occurs. A few examples may help clarifying this point:

```
result :: Int
result = read "1" + 2
```

In this case the application **read** `"1"` occurs in a context where it must have type **Int**, because `result` has been declared of type **Int**, and since `result` is the sum of two operands, both operands must have type **Int**. Indeed we have:

```
> result
3
```

If we try to evaluate **read** `"1"` in `GHCi`, in a context where `GHCi` has no information on which type **read** `"1"` must have, we get an error:

```
> read "1"
<interactive>:1:0:
    Ambiguous type variable ‘a’ in the constraint:
      ‘Read a’ arising from a use of ‘read’ at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

GHCi complains that there is an "ambiguous type variable", which is the one arising from the type of **read**. This type variable must belong to the class **Read**, but there is no way to enforce this fact. If we expect GHCi to give a sensible answer, we must provide enough context information either by means of an explicit type declaration as we have done with result above, or simply by forcing **read** "1" to have a specific type:

```
> read "1" :: Int
1
> read "1" :: Float
1.0
```

10.6 Case study: showing and reading hexadecimal numbers

It is instructive to implement specialized versions of **show** and **read** for showing and reading hexadecimal numbers. This will give us the chance to put together many concepts we introduced so far. Let us start with implementing a function showHexInt which, in our context, should convert an integer number to a string of hexadecimal digits, namely to a string of characters between '0' and '9' and between 'A' and 'B'. In the development that follows we will crucially rely on the fact that the characters corresponding to the decimal digit and those corresponding to the letter of the latin alphabet are numbered in order. For example, '0' is numbered 48, '1' is numbered 49, up to '9' that is numbered 57. Similarly, 'A' is numbered 65, 'B' is numbered 66, and so on. Although the exact numbers do not matter – we will carefully avoid to mention them explicitly in our code – it is important to keep in mind this correspondence.

Let us start with a simple task, that of converting an integer number included between 0 and 15 to a hexadecimal digit:

```
import Data.Char

showHexDigit :: Int → String
showHexDigit d | d >= 0 && d < 10 = [chr $ ord '0' + d]
               | d >= 10 && d < 16 = [chr $ ord 'A' + d - 10]
```

We have two cases: if the number d is comprised between 0 and 9, then it is converted to a decimal digit by finding out the d-th character after '0'; if the number d is comprised between 10 and 15, then it is converted to a hexadecimal digit by finding out the (d - 10)-th character after 'A'. Before proceeding further, let us make sure that showHexDigit is correct:

```
> showHexDigit 5
"5"
> showHexDigit 10
"A"
> showHexDigit 15
"F"
```

Observe that showHexDigit is a partial function, it is not defined for numbers that are not in the range between 0 and 15. This is not a problem since in what follows we will apply it only to numbers within this range.

The conversion of an integer number x to a hexadecimal number is implemented as a recursive function whose base case occurs when x is within the range [0..15]. In this case it suffices to apply the showHexDigit function we have already implemented to x. If x is greater than 15, then we split it into two smaller numbers y and r such that $x = 16y + r$ where $0 \leq r \leq 15$. Then, the hexadecimal representation of x is given by the hexadecimal representation of y followed by the hexadecimal representation of r. Since r lies in the range [0..15], we can apply showHexDigit on it. Since y is smaller than x, we can recursively apply the conversion to it. Then, it suffices to concatenate the two results together:

```
showHexInt :: Int → String
showHexInt x | x >= 0 && x <= 15 = showHexDigit x
             | otherwise = showHexInt (x ‘div‘ 16)
                           ++ showHexDigit (x ‘mod‘ 16)
```

Let us try showHexInt a few times to be sure that it does what we want:

```
> showHexInt 256
"100"
> showHexInt 1234
"4D2"
> showHexInt 65535
"FFFF"
```

Not surprisingly, the conversion from a string of hexadecimal digits to an integer number is slightly more complicated. As we did in the case of showHexInt, we start by solving a simpler problem, that of converting a single hexadecimal digit to an integer number in the range [0..15]. To add some flexibility, we extend the conversion so that we are also able to deal with lowcase letters between 'a' and 'f', in addition to the uppercase ones:

```
readHexDigit d | d >= '0' && d <= '9' = ord d - ord '0'
               | d >= 'A' && d <= 'F' = ord d - ord 'A' + 10
               | d >= 'a' && d <= 'f' = ord d - ord 'a' + 10
```

There is really nothing special about readHexDigit as it simply performs the inverse function of showHexDigit:

```
> readHexDigit '5'
5
> readHexDigit 'A'
10
> readHexDigit 'F'
15
> readHexDigit 'f'
15
```

For the general conversion, we should keep in mind that Haskell strings are lists of characters and thus the most natural way of scanning strings is from left to right. This means that we will scan through the hexadecimal number starting from the most significant digit. With these insights, an interative algorithm for computing the integer number represented by a string of hexadecimal digits is the following:

1. set x to 0

2. if the string is finished, then the result is x
3. set c to the next character in the string
4. set x to `16 * x + readHexDigit c`
5. go to step 3

By recalling the discussion in Section 6.2, where we have seen how to implement apparently imperative algorithms such as this one in a purely functional style, we can provide our own implementation of the conversion function by defining an auxiliary function that has two arguments: one is the x being assigned in the algorithm above; the other is the remainder of the string to be examined:

```
readHexInt :: String → Int
readHexInt s = aux 0 s
    where
      aux :: Int → String → Int
      aux x "" = x
      aux x (c : cs) = aux (16 * x + readHexDigit c) cs
```

Observe that all the work is performed within `aux`. The function `readHexDigit` simply starts the iteration (it corresponds to step 1 in the above algorithm, where x is set to 0). We conclude this section with a few tests for `readHexInt`:

```
> readHexInt "100"
256
> readHexInt "4d2"
1234
> readHexInt "ffff"
65535
```

10.7 Exercises

Exercise 10.1. Re-implement the function `myShowList` in Example 10.1 using the `Data.List.intersperse` library function.

Exercise 10.2. Modify the function `readHexInt` defined in Section 10.6 so that it correctly converts strings of hexadecimal digits possibly preceeded and/or followed by spaces.

Exercise 10.3. The library function **`words`** takes a string s and returns the list of "words" in s, where a word is any nonempty sequence of characters other than spaces. Without looking at the source code of the standard library, implement your own version of **`words`**.

Exercise 10.4. Define the inverse function of **`words`** (see Exercise 10.3), namely a function that, when applied to a list of strings, produces a string obtained by concatenating all the strings in the list into a single string, separating the strings with a single space. Make sure that no space is added in front of the first string and after the last string in the list.

Exercise 10.5 (*). Define a function to convert a string into a fractional number. Assume that the string is made of decimal digits only, and possibly one occurrence of the `.` character.

Chapter 11

Pattern matching

11.1 Patterns

A pattern is a syntactic construct used to specify a collection of related values and to provide names for accessing parts of such values. It can be thought of as a filter through which values are passed: some values are "captured", or *matched*, by the pattern, others are not. In the previous chapters we have seen quite a few examples of patterns: `0`, which matches the number 0; `_`, which matches any value; `n`, which also matches any value, and in addition binds the name `n` to the matched value; `[]`, which matches the empty list; `x : xs`, which matches a nonempty list and additionally binds the name `x` to the head of the list and the name `xs` to the tail of the list.

More formally, we can describe patterns as follows:

- every literal (such as `0`, `1.5`, `'a'`) is a pattern;
- `_` is a pattern;
- every identifier is a pattern;
- `[]` is a pattern;
- if P_1 and P_2 are patterns, then P_1 `:` P_2 is a pattern;
- if $P_1, P_2, \dots, P_n$ are patterns (with $n > 0$), then `(`P_1`, `P_2`, ..., `P_n`)` is a pattern.

and we will see a few more formation rules for patterns in the following.

Recall that strings are just lists of characters. Hence the string `"Haskell"` is nothing but

```
'H' : 'a' : 's' : 'k' : 'e' : 'l' : 'l' : []
```

namely `"Haskell"` is also a pattern according to the rules above.

Patterns must obey a *linearity condition* which roughly states that the same variable must not occur twice within the same pattern. For example, `_`, `x`, and `(x, y)` all satisfy the linearity condition, whereas `(x, x)` and `x : _ : x` do not. A nonlinear pattern such as `(x, x)` may in fact specify a useful set of values. In this case, the set of pairs whose components are equal. However, the pattern does not say explicitly what kind of equality is intended and, in general, non linear patterns are difficult to implement efficiently. For these reasons, Haskell forbids nonlinear patterns. This restriction is not a real limitation, though, since equality constraints can be easily enforced by means of guards. As an example, the effect of the nonlinear pattern (x, x) could be obtained by means of the equation

```
f (x, y) | x == y = ...
```

Just like expressions, patterns do have a type, which is the type of the values they match. For instance, the pattern 0 has type **Num** t ⇒ t, the pattern _ has type 'a', the pattern [] has type [a], and so on and so forth. When a function is defined by pattern matching, the patterns corresponding to the same argument must all have the same type which must be consistent with the type of the argument.

11.2 Exhaustiveness

Pattern matching provides the programmer with a tool that simultaneously accomplishes two distinct tasks: the first is to define a function conditionally, depending on the structure of the arguments of the function; the second is to possibly bind some values from the arguments being matched, so that it is possible to refer to them in the body of the function. In Chapter 9 we have seen a definition of the function **length** that uses pattern matching:

```
length :: [a] → Int
length [] = 0
length (_ : xs) = 1 + length xs
```

The function is defined by two distinct equations. The first one applies only when the argument of **length** is [], the empty list. The second equation applies only when the argument of **length** has the form _ : xs. This pattern denotes a nonempty list (see the : constructor). It does not bind any name corresponding to the head of the list, since this value is never used in the body of the function on the right hand side of =, but it does bind the name xs to the tail of the list being matched.

The pattern matching defined by the two equations above is *exhaustive*, because overall it covers up every possibility. When applying **length** to a list, the list is either empty, in which case the first equation applies, or it has at least one element, in which case the second equation applies. In other terms, the two patterns [] and _ : xs taken together are able to capture any list. Exhaustiveness is a fundamental property because it ensures that in no case the application of **length** can fail because of an "unexpected value". The domain of **length**, [a], tells us that it can be applied to lists only, and the exhaustiveness of pattern matching tells us that there is an equation for every list **length** can be applied to.

In contrast to this, the pattern matching in the functions **head** and **tail** is not exhaustive:

```
head :: [a] → a
head (x : _) = x

tail :: [a] → [a]
tail (_ : xs) = xs
```

We see that in each function there is only one equation whose pattern matches nonempty lists only. There is no equation saying what the behavior is if **head** or **tail** is applied to the empty list. For this reason **head** and **tail** are *partial functions*: when applied to the empty list, they terminate abnormally by raising an exception.

Patterns _ and x for every identifier x are special because they match any value. When we define a function such as

```
successor :: Int → Int
successor n = 1 + n
```

we are using a pattern `n` that matches every possible value that `successor` can be applied to. This makes the pattern matching in the definition of `successor` and other similar functions trivially exhaustive, because the pattern therein does not impose any constraint on the argument.

Exhaustiveness is a property of a whole pattern matching definition, not necessarily of single patterns. The pattern matching in the definition of `successor` and **`length`** is exhaustive, but the pattern `[]` in the first equation of **`length`** would not make an exhaustive pattern matching. A pattern that is exhaustive by itself is called *irrefutable*, since there is no value that is not matched by it. Examples of irrefutable patterns are `_` and `n` for every `n`, but also `(_, _)` since every value of type `(a, b)` is matched by `(_, _)`.

When pattern matching uses *guards*, exhaustiveness is more challenging to determine. For example, consider the following variant of the `absolute` function:

```
absolute :: Int → Int
absolute n | n >= 0 = n
absolute n | n < 0 = negate n
```

This definition is clearly correct, and the pattern matching is exhaustive, but this fact cannot be easily inferred automatically by `GHCi`. The point is that the boolean expressions used in the guards can be arbitrarily complex, and the problem of verifying whether their disjunction is a tautology is undecidable in general. Haskell adopts a pragmatic approach: a guard is exhaustive if its final branch is tagged with **`otherwise`**, and exhaustiveness of pattern matching is checked by looking only at those patterns with exhaustive guards (a missing guard is trivially exhaustive, since it is equivalent to **`otherwise`**). Hence, the pattern matching in the definition of `absolute` above is *not* considered exhaustive by Haskell, but the one in

```
absolute :: Int → Int
absolute n | n >= 0 = n
absolute n | otherwise = negate n
```

is exhaustive.

It is possible to ask `GHCi` to inform the programmer when the pattern matching in a function definition is not exhaustive, by enabling the option `-fwarn-incomplete-patterns`:

```
> :set -fwarn-incomplete-patterns
> :load head_tail.hs
[1 of 1] Compiling Main             ( head_tail.hs, interpreted )
head_tail.hs:2:0:
    Warning: Pattern match(es) are non-exhaustive
             In the definition of ‘head’: Patterns not matched: []
head_tail.hs:5:0:
    Warning: Pattern match(es) are non-exhaustive
             In the definition of ‘tail’: Patterns not matched: []
Ok, modules loaded: Main.
```

`GHCi` informs us not only that the pattern matching in those functions is not exhaustive, but it also gives us an example of value that is not matched (`[]` in this case).

11.3 Overlapping patterns and first-match policy

Two patterns are *overlapping* if there is a value that is matched by both. For example, the patterns `0` and `_` are overlapping because the value `0` is matched by both. Also, `_` and `n` are overlapping because they both match every value. The patterns `[]` and `_ : _` are *not* overlapping because the first one only matches the empty list, whereas the second one only matches nonempty lists.

Overlapping patterns are a potential source of ambiguity in the definition of functions by pattern matching. The definition of **`length`** above is not ambiguous because the patterns on the left hand side of the equations are not overlapping. Thus, given a list l, there is always exactly one equation that applies. Conversely, the definition of `safeRecip`

```
safeRecip :: (Fractional a) ⇒ a → a
safeRecip 0 = 0
safeRecip n = 1 / n
```

makes use of the two overlapping patterns `0` and `n`. In this case, the application `safeRecip 0` is potentially ambiguous, since both equations can be used. To avoid this kind of ambiguities, Haskell adopts a so-called *first-match policy* for pattern matching. When multiple equations apply, the one that occurs first (from top to bottom) is the one chosen. With this policy, the application `safeRecip 0` is no longer ambiguous, since the pattern `0` matches the value `0`, hence the first equation in the definition of `safeRecip` is used.

When guards are present, the first-match policy mandates that if multiple equations apply, then the one that occurs first (from top to bottom) and whose guard evaluates to **`True`** is the one chosen.

11.4 Ascriptions

Consider the following function `sortPair`, that takes a pair of values and produces another pair where the values have been sorted with the smaller one in the first component and the larger one in the second component:

```
sortPair :: (Ord a) ⇒ (a, a) → (a, a)
sortPair (x, y) | x <= y = (x, y)
                | otherwise = (y, x)
```

In order to be able to compare the two components `x` and `y` of the original pair, it is necessary to use the pattern `(x, y)`. However, if the two components of the pair are already sorted, a *new* pair `(x, y)` is created on the right hand side of `=`, which is equal to the original one. Hence, the original pair is first deconstructed by pattern matching, only to be constructed again in the first case. In this case, it would be convenient to have *both* the pattern `(x, y)` so that the components of the original pair can be accessed, *and also* a binding `p` for the whole pair, so that if we need to return it unchanged we can just write `p` without requiring further memory allocations. This is possible in Haskell using *ascriptions*. An ascription is a pattern of the form x@p where x is an identifier and p is a pattern. The ascription matches all the values matched by p but, in case of match, it also binds the name x to the value being matched. In the right hand side of the rule, one can thus access both the components of the value as specified by p, but also the whole value by means of the name p. The function `sortPair` can be rewritten using ascriptions as follows:

```
sortPair :: (Ord a) ⇒ (a, a) → (a, a)
sortPair p@(x, y) | x <= y = p
                  | otherwise (y, x)
```

Names used in ascriptions must obey the usual linearity constraints of patterns. In particular the name x in an ascription x@p cannot occur within p.

11.5 Exercises

Exercise 11.1. Infer the most general type of the following functions and check your answers with GHCi:

1.
```
f x y z = x == y && y == z
```

2.
```
f x _ y = x == y
```

3.
```
f x y _ | x < y = x
f _ y z | y < z = y
```

4.
```
f [] _ = 0
f _ [] = 0
f (_ : xs) (_ : ys) = 1 + f xs ys
```

5.
```
f [] = 0
f [(_, y)] = y
f ((x, y) : _) = x
```

6.
```
f [] = []
f (p : ps) = fst p : f ps
```

7.
```
f "" = True
f _ = False
```

8.
```
f [] = True
f (c : _) | Data.Char.isSpace c = False
f (_ : cs) = f cs
```

Exercise 11.2. Determine which of the following function definitions are well formed and well typed. For those that are, say which are exhaustive and provide their most general type.

1.
```
f x y = (x, y)
```

2.
```
f x y = (x, x)
```

3.
```
f [] = []
f x@(y : ys) = x : y
```

4.
```
f [] = []
f x@(y : ys) = x ++ ys
```

5.
```
f False x _ = x
f True _ x = x
```

6.
```
f (x, y) = x : y
f [x, y] = x : y
```

7.
```
f (x, y) = x : y
f (x : y) = x : y
```

8.
```
f ((_, _, x) : _) = x
f [_] = 0
```

Part IV

Function transformations

Chapter 12

First-class functions

The essence of functional programming is being able to treat functions like any other value. This capability enables powerful forms of abstraction that allow the programmer to build general libraries of functions implementing recurring patterns of computation and to easily compose and specialize functions for solving specific problem instances. In this chapter we will enter the world of first-class functions and see how they can be used for improving modularity and code reuse.

12.1 Functions as first-class values

In a functional programming language there is no conceptual difference between functions and data. Functions can be treated as data, in particular they can be passed as arguments to other functions, they can be returned as results of functions, they can be stored within tuples, lists, and any other (user-defined) data structure. The fact that functions are first-class values in Haskell is indirectly implied by the observation that they have a type, just as any other value does. But this hardly makes any justice to all the implications that first-class functions have. To better illustrate the point, consider the function `sinCos` below that takes a boolean value `b` and a floating-point number `x` as arguments and computes the sine of `x` if `b` is **`True`** and the cosine of `x` otherwise. The obvious way to write this function is:

```
sinCos :: Bool → Float → Float
sinCos b x = if b then sin x else cos x
```

In the example **`sin`** and **`cos`** are functions, but they are immediately applied to the argument `x`, so this implementation somehow assumes that functions do not make sense *per se*, but only when they are applied to their arguments. If this version of `sinCos` is unimpressive, the following one, which is entirely equivalent, is much more interesting:

```
sinCos :: Bool → Float → Float
sinCos b x = (if b then sin else cos) x
```

Here we have a sub-expression, **`if`** `b` **`then sin else cos`**, whose evaluation yields either the function **`sin`** or the function **`cos`** depending on the value of `b`. Only then the function resulting from this evaluation, which is not statically known by looking at the code, is applied to the argument `x`.

An even stronger evidence of the fact that functions and data (say, numbers) are equally supported in Haskell comes from the ability of defining functions “on the fly”, just where we need them:

```
> (\x → x + 1) 2
3
```

Here we have defined the function `(\x → x + 1)` and have immediately applied it to the argument `2`. The function thus defined has one argument `x` (on the left hand side of `→`) and a body `x + 1`. Namely, the function accepts `x` and returns the successor of `x` pretty much as `successor` we have defined previously, except that we have defined it on the spot, *and we haven't given it a name*: the function `(\x → x + 1)` is *anonymous*.[1]

The reduction of an anonymus function proceeds as for named functions:

```
  (\x → x + 1) 2  ⟹  2 + 1  ⟹  3
```

namely, the argument is replaced by the expression next to the function. Observe that `(\x → x + 1)` is a well-formed, well-typed Haskell expression of its own. It needs not be applied to an argument to make sense. In fact, we can ask `GHCi` about its type, and obtain:

```
> :type \x → x + 1
\x → x + 1 :: (Num a) ⇒ a → a
```

The type inferred by `GHCi` is more generic than **`Int`** `→` **`Int`** we have explicitly provided for `successor`. This is because the function is only defined in terms of `1`, which is of type `(`**`Num`** `t) ⇒ t`, and `(+)`, which is of type `(`**`Num`** `a) ⇒ a → a → a`.

Also, observe that in `(\x → x + 1) 2` the parentheses are necessary since, according to Haskell's precedence rules, the body of a function extends to the right as much as possible.

12.2 The true nature of multi-argument functions

In Chapter 4 we have seen several ways to give a name to a value. Now we realize that the mechanism for defining functions that we have been using so far is just a special case where we give a name to a value. Indeed, we can re-define the successor function as follows:

```
successor :: Int → Int
successor = \x → x + 1
```

Compared to the previous definition of `successor`, here we have "shifted" the argument of `successor` to the right hand side of `=` and turned it into the argument of the anonymous function `\x → x + 1`. So, the previous definition of `successor` read "`successor` applied to an argument `x` is `x + 1`" whereas this definition of `successor` reads "`successor` is a function with argument `x` and body `x + 1`". But these two ways of interpreting `successor` are just the same.

We can do the same "shifting" with multi-argument functions. Let us consider the first definition of `divides` we have introduced:

```
divides :: Int → Int → Bool
divides m n = m /= 0 && n ‘mod‘ m == 0
```

and let us shift the `n` argument to the right hand side of the equation:

```
divides :: Int → Int → Bool
divides m = \n → m /= 0 && n ‘mod‘ m == 0
```

[1]The symbol \ in front of the argument is a rough approximation of the greek letter λ, which characterizes the first functional language ever designed, the λ-calculus, developed by Alonzo Church in the 30's. In λ-calculus, the successor function would be written as $\lambda x.x + 1$.

This definition is correct and totally equivalent to the previous one; it reads: “`divides` applied to `m` is a function with argument `n` that returns `m /= 0 && n ‘mod‘ m == 0`.” But we can actually do more and shift the first argument as well:

```
divides :: Int → Int → Bool
divides = \m → \n → m /= 0 && n ‘mod‘ m == 0
```

which reads: “`divides` is a function with argument `m` that returns a function with argument `n` that returns `m /= 0 && n ‘mod‘ m == 0`.”

Note that in both shiftings the type of `divides` has stayed the same as in the original definition, despite the way we read the latter two definitions has changed dramatically: we are now claiming that `divides` is a function returning a function! In order to solve this mystery, we must now reveal that in Haskell there is no such thing as a multi-argument function. The notation

$f\ x_1\ x_2\ \cdots\ x_n\ =\ E$

that we have been using for defining n-ary functions is just a shorthand for

$f\ =\ \backslash x_1\ \rightarrow\ \backslash x_2\ \rightarrow\ \cdots\ \backslash x_n\ \rightarrow\ E$

and symmetrically the notation

$f\ E_1\ E_2\ \cdots\ E_n$

that we have been using for applying function f to the arguments $E_1, E_2, \ldots, E_n$ is just a shorthand for

$(\cdots((f\ E_1)\ E_2)\ \cdots\ E_n)$

namely function application is a binary, left-associative, invisible operator.

At the type level, → is a *binary type-constructor*: it takes two types S and T and produces another type $S\ \rightarrow\ T$ which is the type of those functions that accept values of type S and return values of type T. For example:

- `Int → (Int → Bool)` is the type of functions that accept integer numbers and return a function that accepts integer numbers and returns a boolean value;

- `(Int → Int) → Bool` is the type of functions that accept a function – accepting integer numbers and returning integer numbers – and return a boolean value.

Furthemore, the → type constructor is *right associative*. That is $S_1\ \rightarrow\ S_2\ \rightarrow\ T$ means $S_1\ \rightarrow\ (S_2\ \rightarrow\ T)$. So, `divides` has always been a function returning a function, since its very first definition.

To convince ourselves of these facts, let us apply the latest version of `divides` to two arguments, but one at a time:

```
> ((divides 2) 4)
True
> ((divides 0) 1)
False
> ((divides 2) 3)
False
```

If we take the first expression, it is reduced thus:

```
((divides 2) 4) ⟹ (((\m → \n → m /= 0 && n ‘mod‘ m == 0) 2) 4)
                ⟹ ((\n → 2 /= 0 && n ‘mod‘ 2 == 0) 4)
                ⟹ 2 /= 0 && 4 ‘mod‘ 2 == 0
                ⟹ True
```

In summary, every Haskell function has exactly one argument and multi-argument functions are implemented as a cascade of functions returning functions. The illusion that some functions have multiple arguments results from the combination of two dual conventions: function application is a binary operator that associates to the left, and the arrow type → is a binary type constructor that associates to the right. It is still useful, at an informal level, to reason about the "number" of arguments of a function, but we should be aware, from now on, that this number may be inaccurate depending on the point of view. We will see examples of this inaccuracy just in the next section, where we develop functions that create other functions.

12.3 Function composition and application

Being functions first-class values it is trivial to write functions that transform functions. In mathematics the perhaps most common example of function transformation is the *function composition* operator: $(f \circ g)(x) = f(g(x))$. That is $f \circ g$ is the functional composition of f and g. In Haskell we define a function `compose` like this:

```
compose :: (b → c) → (a → b) → a → c
compose f g = \x → f (g x)
```

From its definition it is clear that `compose` takes two arguments `f` and `g` and returns a function that accepts an argument `x` and returns `f (g x)`. The type of `compose`, while slightly confusing at first, can be intuitively explained thus: if we think of functions as of transformation boxes, we see that `g`, which has type `a → b` is the first transformation that is applied, turning values of type `a` into values of type `b`. Then, `f` is applied. Clearly, the codomain of `g` and the domain of `f` must coincide for this composition to be well typed. Thus, `f` needs to have type `b → c` for an arbitrary type `c`. Finally, recall that the arrow type operator → is right associative, so the type of `compose` can be rewritten as

```
  (b → c) → (a → b) → (a → c)
```

namely, `compose f g` is a function of type `a → c`, having as domain the domain of `g` (the first transformation that is applied) and having as codomain the one of `f` (the second transformation that is applied).

The Haskell standard library defines a function `(.)` that behaves just as `compose`:

```
> :type (.)
(.) :: (b → c) → (a → b) → a → c
```

Using `(.)` it is possible to express in a very compact and convenient many useful functions, of which we give just a few examples here:

- **`not`** `.` **`isSpace`** is the function that applied to a character `c` returns **`True`** if `c` is *not* a space, and **`False`** otherwise;
- `(\x → x ^ 2) .` **`sin`** is the function $\sin^2$;
- **`head`** `.` **`tail`** is the function that returns the second element of a list.

Example 12.1. Let us define a function `nth` that takes a function f and a number n and returns f^n, namely $f \circ f \circ \cdots \circ f$ where there are n occurrences of f. We implement `nth` as a recursive function: in the base case ($n = 0$) we do not compose any f. If we think of f as of a transformation box, not applying any f means returning the function that performs no transformation, that is the identity function **id**. In the recursive case ($n > 1$), it suffices to compose f once with $n - 1$ compositions of f. In Haskell code:

```
nth :: Int → (a → a) → a → a
nth 0 _ = id
nth n f = f . nth (n - 1) f
```

Observe from its type that `nth` accepts an integer number and a function of type `a → a` and returns a function of type `a → a`. ■

Just as it's possible to express Haskell function composition in Haskell, it is as well possible to express Haskell function application. Here is an `apply` function that does just that:

```
apply :: (a → b) → a → b
apply f x = f x
```

Here the function `apply` takes two arguments, a function of type `a → b` and a value of type `a`, and returns a value of type `b` which is obtained by applying the function to the value. But this is not the only way one can define `apply`, here is an alternative implementation:

```
apply :: (a → b) → a → b
apply f = f
```

Instead of defining `apply` as a two-arguments function, we define `apply` as a single argument function of type `(a → b) → (a → b)`. The argument of `apply` is itself a function of type `a → b` and `apply` returns the same function. But then we realize that `apply` looks very much like the identity function, except for the type, that is more specific than that of **id**, and for the name of the function, which is irrelevant as far as the behavior of the function is concerned. Thus, we can ultimately implement `apply` as follows:

```
apply :: (a → b) → a → b
apply = id
```

In a language with first-class functions, and with functions that return functions in particular, the concept of "number of arguments" of a function may change depending on the point of view. We have defined three equivalent variants of `apply`, the first with two arguments, the second with one argument only, and the third with no arguments at all!

By the way, `apply` is not just a contrivied example, but is a practically useful function defined in the Haskell standard library, where it is called `($)`. Since the operator `$` in Haskell is given a very low precedence, when used in infix form, this function can save writing some parentheses. Thus we can write, for example:

```
> sin $ pi / 2
1.0
```

instead of

```
> sin (pi / 2)
1.0
```

12.4 Case study: numeric integration

Let f be a (mathematical) function. Suppose we want to compute the definite integral of f in the interval $[a..b]$, which we assume being enclosed within f's domain. It is well known that, intuitively, computing the integral of a function corresponds to computing the area on the cartesian plane determined by the x-axis, the limits a and b, and the function f. Using this intuition, we can approximate the integral of f from a to $a+\delta$ with the area of the rectangle whose base is δ and whose height is $f(a)$:

$$\int_a^{a+\delta} f(x)dx \approx \delta f(a)$$

The approximation is more and more precise as δ gets smaller. We can then compute the numeric integral of f between a and b by dividing the interval $[a..b]$ in n sub-intervals of equal width, by computing the approximate numeric integral for each sub-interval, and finally by summing up all the approximate intergrals:

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=0}^{n-1} \int_{a+i\delta}^{a+(i+1)\delta} f(x)dx \\ &\approx \sum_{i=0}^{n-1} \delta f(a+i\delta) \\ &= \delta \sum_{i=0}^{n-1} f(a+i\delta)\end{aligned}$$

where

$$\delta = \frac{b-a}{n}$$

This approximation of the numeric integral depends on 3 parameters: the function f being integrated, the interval $[a..b]$ of integration, and the number n of sub-intervals. Thus it makes a perfect case for illustrating a concrete application of higher-order functions. In Haskell we implement this approximation of numeric integration thus:

```
integrate :: (Float → Float) → Float → Float → Int → Float
integrate f a b n = delta * s
  where
    delta = (b - a) / fromIntegral n
    s = sum [ f (a + delta * fromIntegral i) | i ← [0 .. n - 1] ]
```

where we use the **fromIntegral** library function to convert an integer number to a floating-point one.

By integrating the sine function in the interval $[0, \pi]$ with increasing values of `n` we see the result converging to the exact answer:

```
> integrate sin 0 pi 2
1.5707964
> integrate sin 0 pi 20
1.9958858
> integrate sin 0 pi 200
1.9999582
> integrate sin 0 pi 2000
1.9999977
> integrate sin 0 pi 20000
2.0000012
```

12.5 Case study: a toolkit for list manipulation

The possibility of defining higher-order functions, and in particular functions that accept functions as arguments, allows the programmer to define *patterns of computation* as simple Haskell functions. Consider for example the functions mapSucc (Section 9.5) and pyramid' (Example 9.6). In the first case we compute the successor of every element of a list of integer numbers. In the second case we compute the square of every element of a list of integer numbers. The common pattern of these two examples is that we start from a list and create a new list whose elements are obtained by transforming the corresponding element in the original list. Every element is transformed in the same way. By abstracting over the transformation, we can define a higher-order function **map** that embodies this particular way of manipulating lists:

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x : xs) = f x : map f xs
```

With **map** (which is a predefined function in the Haskell **Prelude**), we can implement mapSucc thus:

```
mapSucc :: [Int] → [Int]
mapSucc xs = map (\x → x + 1) xs
```

and pyramid' thus:

```
pyramid' :: Int → Int
pyramid' n = sumInt $ map (\x → x * x) [1..n]
```

Similarly, we can convert a string to upper case by applying the function Data.**Char**.**toUpper** to every character in the string:

```
> map Data.Char.toUpper "Hello,␣World!"
"HELLO,␣WORLD!"
```

In all cases we reuse the **map** function, which implements a generic, uniform list transformation, and we provide it with the transformation that suits our needs depending on the context. Sometimes, as for Data.**Char**.**toUpper**, the transformation is a function already available. In other cases it is not, but we need not bother defining a specific function, since we can provide one on-the-fly, as we did for the successor \x → x + 1 and the square function \x → x * x.

Consider now the functions filterEven and divisors (Section ??). In both cases we need to remove those elements of a list that do not satisfy a specific predicate. By abstracting over the predicate, we can define the following higher-order function **filter**, which we have already used in the introductory Quick Sort example:

```
filter :: (a → Bool) → [a] → [a]
filter _ [] = []
filter p (x : xs) | p x = x : filter p xs
                  | otherwise = filter p xs
```

With **filter** we can implement filterEven thus:

```
filterEven :: [Int] → [Int]
filterEven xs = filter even xs
```

and divisors thus:

```
divisors :: Int → [Int]
divisors n = filter (\x → x ‘divides‘ n) [1..n]
```

Again, we have abstracted the filtering of elements into a function **filter** and have provided it the right filter according to the specific problem we are tackling.

Finally, there are cases where one needs to combine all the elements of a list. This is the case of sumInt and prodInt, where all the elements of a list are respectively added or multiplied together. Roughly speaking, supposing that

$[x_1, x_2, \ldots, x_n]$

is a list, what we want to do is to compute a value

$x_1 \oplus x_2 \oplus \cdots \oplus x_n$

for some operator $\oplus$. In sumInt the operator is +, whereas in prodInt the operator is $*$. However, this intuition is not completely specified, since it is not clear what the result would be if the original list is empty. Furthermore, the operator $\oplus$ is not necessarily associative (as + and $*$), so the order in which we perform the reductions of the above expression may matter. This is why there are two different ways of reducing a list to a value. If we reduce the list *from the left* we compute

$(\cdots((a \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n$

whereas if we reduce the list *from the right* we compute

$x_1 \oplus (x_2 \oplus (\cdots \oplus (x_n \oplus a)\cdots))$

where we have indicated with a the value to be used in the base case. Corresponding to these two reductions we can define the **foldl** and **foldr** functions as follows:

```
foldl :: (a → b → a) → a → [b] → a
foldl _ a [] = a
foldl f a (x : xs) = foldl (a ‘f‘ x) xs

foldr :: (a → b → b) → b → [a] → b
foldr _ a [] = a
foldr f a (x : xs) = x ‘f‘ foldr f a xs
```

With **foldl** (or **foldr**) we can implement sumInt and prodInt thus:

```
sumInt :: [Int] → Int
sumInt xs = foldl (+) 0 xs

prodInt :: [Int] → Int
prodInt xs = foldl (*) 1 xs
```

Observe that in this case using **foldl** or **foldr** does not make any difference, since both + and $*$ are associative.

As another example, here is another way of computing the length of a list:

```
length :: [a] → Int
length xs = foldr (const succ) 0 xs
```

Example 12.2 (efficient reverse). In Example 9.5 we have seen an intuitive implementation of the function that reverses the elements of a list. We have argued that rev must necessarily make use of ++, since it needs to append elements *at the end* of a list, rather than at the beginning. This makes rev quite inefficient, since appending at the end of a list has a cost that is proportional to the length of the list.

A cleverer way of implementing rev stems from the left folding of a list of elements:

$(\cdots((a \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n$

Suppose that ⊕ is an operator that prepends (with `:`) its right operand to its left operant. Namely, ⊕ is the symmetric version of `:`. Let us call this operator "snoc" (the reverse of "cons"):

```
snoc :: [a] → a → [a]
snoc xs x = x : xs
```

We realize that the left folding of a list where ⊕ is `snoc` reduces as follows:

$$
\begin{aligned}
&(\cdots((a \text{ `snoc` } x_1) \text{ `snoc` } x_2) \text{ `snoc` } \cdots) \text{ `snoc` } x_n\\
&\Longrightarrow (\cdots((x_1 : a) \text{ `snoc` } x_2) \text{ `snoc` } \cdots) \text{ `snoc` } x_n\\
&\Longrightarrow (\cdots(x_2 : x_1 : a) \text{ `snoc` } \cdots) \text{ `snoc` } x_n\\
&\Longrightarrow \cdots \Longrightarrow x_n : \cdots x_2 : x_1 : a
\end{aligned}
$$

Hence we can implement a linear-time `rev` as a specialization of `fold`, where the folding function is `snoc` and the base element a is `[]`:

```
rev :: [a] → [a]
rev xs = foldl snoc [] xs
```

■

12.6 Exercises

Exercise 12.1. Explain the apparent absurdity arising from the last definition of `apply`, where clearly `apply` is a function and yet no argument occurs in the equation.

Exercise 12.2. Define a function `makeList` that, when applied to a function f of type `Int → a` and to an integer number n, creates a list of length n whose first element is given by f `0`, whose second element is given by f `1`, up to the last element that is given by f `(n - 1)`.

Exercise 12.3. Make sure to solve Exercise 12.2 using **`map`**.

Exercise 12.4. Define a function **`takeWhile`** that, when applied to a predicate p and to a list l, returns the longest prefix of l whose elements satisfy p.

Exercise 12.5. Define a *non-recursive* function that, when applied to a list of integer numbers, returns **`True`** if all the numbers are even, **`False`** otherwise.

Exercise 12.6. Define a *non-recursive* function that, when applied to a list of pairs whose first component is an integer number, returns **`True`** if all the first components of the pairs are even, **`False`** otherwise.

Exercise 12.7. Define a *non-recursive* function that removes the last element of a list l, namely that returns the longest prefix of l that is different from l.

Exercise 12.8. Define a *non-recursive* function that removes every element of a list that is in an odd position (assume that the first element is in position 1).

Exercise 12.9. Define a *non-recursive* function to check whether a list is palyndrome.

Exercise 12.10. Define a *non-recursive* function that takes a key k and an association list l and returns the list of all the values associated with k in l.

Exercise 12.11. Define a *non-recursive* function that, when applied to a list l, returns another list whose length is the same as that of l and where each element is the number of occurrences of the corresponding element in l. For example, given the list `[1, 3, 2, 1, 1, 3]` the function must return the list `[3, 2, 1, 3, 3, 2]`.

Exercise 12.12. Locate the functions **`all`** and **`any`** in the Haskell standard library. Provide your own implementation of these two functions, without looking at the source code of the standard library.

Exercise 12.13. Define a *non-recursive* function that, when applied to a list of functions `[`f_1`, ..., `f_n`]`, produces the function f_1 `. ··· .` f_n.

Exercise 12.14 (*). Define **`filter`** as a combination of **`foldl`** (or **`foldr`**) and **`map`**.

Exercise 12.15 (*). Definire una funzione *non ricorsiva* `unpack_assoc` inversa della funzione `pack_assoc` dell'esercizio **??**.

Exercise 12.16 (**). Define a function to compute the permutations of the elements of a list. For example, the function applied to `[1, 2, 3]` must produce (a permutation of) the list

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Chapter 13

Function specialization

In this chapter we explore in some more details all the implications of the way Haskell handles functions. In particular, we realize that encoding multi-argument functions as cascades of one argument functions enables interesting forms of code reuse and specialization. We also introduce *section*, a handy syntactical convenience that allows us, at last, to fully understand the Quick Sort implementation we have given in the first chapter.

13.1 Partial application

In the previous chapter we have discovered that the notion of multi-argument function is inaccurate. In a functional programming language such as Haskell, every function has *exactly* one argument, and has type $S \to T$ where S, the domain, is the type of values accepted by the function and T, the codomain, is the type of values returned by the function. We have seen that the type of `divides` actually is `Int → (Int → Bool)`. Namely, `divides` is a function that accepts *one* argument of type `Int` and returns a function that accepts *one* argument of type `Int` and returns a value of type `Bool`. We can use `GHCi` to see whether that's true:

```
> :type divides 2
divides 2 :: Int → Bool
```

We have applied `divides` to just one argument, and `GHCi` says that `divides 2` is indeed a function of type `Int → Bool`. But *which* function? To find this out, let us compute `divides 2` according to its last definition:

```
  divides 2 ⟹ (\m → \n → m /= 0 && n `mod` m == 0)
            ⟹ \n → 2 /= 0 && n `mod` 2 == 0
```

We have obtained a function that accepts an integer number `n` and returns the value of

```
  2 /= 0 && n `mod` 2 == 0
```

Since `2 /= 0` is trivially true, this is equivalent to

```
  n `mod` 2 == 0
```

But then, the function we have obtained looks very much like `even`. Let us give a name to this function:

```
mystery :: Int → Bool
mystery = divides 2
```

and let us check whether the intuition is correct:

```
> mystery 0
True
> mystery 2
True
> mystery 100
True
> mystery 3
False
```

In conclusion, `mystery` is indeed equivalent to the function **`even`**, but the way we have defined `mystery` is rather peculiar: the only equation for `mystery` has no argument. The equation states that `mystery` is a specialized version of `divides` where we have fixed its first argument to the value `2`.

We can redefine some functions that we have explicitly implemented in previous chapters as specializations of library functions (more interesting examples will follow in the next chapters):

- `(+) 1` is the successor function, equivalent to `successor`;
- `(==) 0` is the function that tests whether a number is zero, equivalent to `iszero`.

We can now appreciate two main advantages in representing multi-argument functions as higher-order functions: the first is a theoretical one, in that the language does not *need* a native concept of multi-argument function. This makes the language simpler, both at the level of expressions and also at the level of types. The second advantage is practical and derives from the possibility of partially applying multi-argument functions. Partial application provides an effective and flexible tool for enhancing code reuse.

Example 13.1. An alternative way of defining the **`recip`** function (Example 5.1) is to specialize `(/)` like this:

```
recip :: (Fractional a) ⇒ a → a
recip = (/) 1
```

In this way we do not need to give a name to **`recip`**'s argument. ■

13.2 Sections

An important limitation of partial application is that it allows the programmer to specify a subset of the arguments accepted by a function, *starting from the leftmost argument.* For instance, we have seen that `divides 2`, which is equivalent to **`even`**, is the predicate "being divisible by `2`". It is not possible, however, to obtain the predicate "divides `10`" by means of partial application, because we would have to fix `divides`'s second argument to `10`, as opposed to the first one. The same problem arises when we want to specialize non-commutative binary operators such as division or relational operators. For example, **`div`** `2` is the function "`2` divided by" as opposed to "divided by `2`", and similarly `(<) 2` means "being larger than `2`" as opposed to the more intuitive "being less than `2`".

One way out of this problem is to write an anonymous function that abstracts over the first argument, so `\m → divides m 10` is the predicate "divides `10`", `\n →` **`div`** `n 2` is the function "divided by `2`", and `\x → x < 2` is the predicate "being less than `2`". However, by resorting to explicit (albeit anonymous) functions we partially loose the conciseness and elegance of function specialization.

Fortunately Haskell comes to the rescue with *sections*. A section is a Haskell expression enclosed within parentheses where there is a binary operator that misses one if its operands. For example

```
(< 2)
```

is a section where the binary operator < has just its right operand, but not the left one. A section is not a “complete” expression that can be immediately evaluated. Rather, Haskell turns it into an anonymous function. So the section above becomes

```
(\x → x < 2)
```

as we can test by asking GHCi about its type:

```
> :type (< 2)
(< 2) :: (Num a, Ord a) ⇒ a → Bool
```

Sections represent a (limited) form of partial application where we are able to write an infix operator missing one, or both of its arguments. In fact, we have already been using sections such as (+) for denoting the function represented by the binary operator +. Since every function can also be used as an infix operator, when its name is enclosed in backticks, we can write sections such as (‘divides‘ 10) standing for the predicate “divides 10” and (‘**div**‘ 2) standing for the function “divided by 2”.

Example 13.2 (Quick Sort). We now possess all the notions for fully understanding the implementation of the Quick Sort algorithm we have given in the introduction, and that we repeat here:

```
qsort :: (Ord a) ⇒ [a] → [a]
qsort [] = []
qsort (x : xs) = qsort l1 ++ [x] ++ qsort l2
    where
      l1 = filter (<= x) xs
      l2 = filter (> x) xs
```

The function qsort is a polymorphic function accepting lists of elements of type a, where ‘a’ is an instance of the **Ord** class. The function performs pattern matching on the list: if the list is empty, then there is nothing to sort and the empty list is returned; if the list is not empty, the first element, bound to the name x, is used as *pivot*. The tail of the list, bound to the name xs, is partitioned into two sublists called l1 and l2. Each sublist is obtained by means of **filter**, specifying that only the elements that are less than or equal to x (for l1) or greater than x (for l2) must be considered. The predicates used in the two applications of **filter** are appropriate sections involving the operators <= and >. After l1 and l2 have been recursively sorted, the final result is obtained by concatenating together every single piece in the appropriate order. ■

13.3 Curried and uncurried functions

The implementation of multi-argument functions by means of cascades of functions returning functions is sometimes called “currying” (from the name of Haskell Curry) and such functions are said to be “in curried form”. An alternative, and apparently more straightforward, way of dealing with multi-argument functions is by using tuples. For example, the function

```
maxPair :: (Ord a) ⇒ (a, a) → a
maxPair (a, b) | a > b = a
               | otherwise = b
```

computes the maximum between two values of an ordered type, whereas

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

is the polymorphic concatenation relation between lists. Having these definitions in place, we would write

```
> maxPair (1, 2)
2
```

for applying `maxPair` to the arguments `1` and `2` and

```
> appendPair ([1, 2, 3], [4, 5, 6])
[1,2,3,4,5,6]
```

for applying `appendPair` to the arguments `[1, 2, 3]` and `[4, 5, 6]`.

Functions that accept multiple arguments in tuples are said to be “in uncurried form”. The way uncurried functions are defined and applied resembles much more closely the syntax used in most programming languages, and the overall effect is just the same. The main drawback of uncurried functions is that they take all of their arguments at once. Hence, uncurried functions cannot be partially applied, as in practice they just happen to have one argument.

Occasionally it is handy to work with uncurried functions, typically in those cases where we already have a tuple containing the arguments to be passed to a function. It is straightforward to write Haskell functions that convert curried functions into uncurried ones, and viceversa:

```
curry :: ((a, b) → c) → a → b → c
curry f = \x → \y → f (x, y)

uncurry :: (a → b → c) → (a, b) → c
uncurry f = \(x, y) → f x y
```

Thus **`uncurry`** `(++)` and **`uncurry max`** are equivalent to `appendPair` and `maxPair` respectively, whereas **`curry fst`** and **`curry snd`** are equivalent to `first` and `second` we have introduced earlier.

Example 13.3. Suppose we need to write a function `map2` that behaves as **`map`**, except that it applies a *binary* function `f` to corresponding elements of two lists to produce a third list. We could write `map2` directly as follows:

```
map2 :: (a → b → c) → [a] → [b] → [c]
map2 _ [] [] = []
map2 f (x : xs) (y : ys) = (f x y) : map2 f xs ys
```

Alternatively, we might split this transformation in two phases: during the first phase, corresponding elements of the two lists are paired together with **`zip`** (see Example 9.8); during the second phase, each pair is trasformed by means of the function `f`. This way we would be able to express `map2` as a suitable modification of **`map`**. There is one little problem though, as `f` is in curried form, hence it accepts its arguments one at a time, whereas in this second version of `map2` we would be applying it to pairs containing the arguments. Thus we can re-implement `map2` like this:

```
map2 :: (a → b → c) → [a] → [b] → [c]
map2 f xs ys = map (uncurry f) $ zip xs ys
```

that is, we apply **map** to the "uncurried" version of `f` and to the list of pairs of corresponding elements from `xs` and `ys`. ■

13.4 η-equivalence

In Chapter 12 we have argued that

```
apply :: (a → b) → a → b
apply f x = f x
```

and

```
apply :: (a → b) → a → b
apply f = f
```

are equivalent ways of defining the same function, and the justification we have given for this equivalence is that we can see `apply` as having either two arguments and returning the first applied to the second, or as having just one argument and returning it unchanged. The first version of `apply` can also be defined thus:

```
apply :: (a → b) → a → b
apply f = \x → f x
```

Now, we see that there are two different equations defining the meaning of `apply f`. One equation states that `apply f = f`, the other equation states that `apply f = \x → f x`, and the two equations must be equivalent! More generally, we have that

E

and

`\x →` *E* `x`

are equivalent expressions, provided that `x` does not occur in *E*. By "equivalent expressions" here we mean that the effect of applying *E* to an argument *F* is exactly the same as applying `\x →` *E* `x` to an argument. Indeed we have

`(\x →` *E* `x)` *F* $\Longrightarrow$ *E F*

(observe that the hypothesis that `x` does not occur in *E* is essential for this reduction to be correct).

We say that `\x →` *E* `x` is the *η-expansion* of *E* and that *E* is the *η-contraction* of `\x →` *E* `x`.[1] Overall we say that *E* and `\x →` *E* `x` are *η-equivalent*.

Example 13.4. In Section 12.5 we have been able to define the function `mapSucc` in terms of a more general function **map**, which accepts a trasformation function to be uniformly applied to all the elements of a list. We have defined `mapSucc` thus:

```
mapSucc :: [Int] → [Int]
mapSucc xs = map (\x → x + 1) xs
```

[1]Apparently it would be more correct to say that `\x → E x` is *one particular* η-expansion of *E* since we may have many different η-expansions of *E* depending on the name `x` we choose. However, the particular name we choose does not change the meaning of the expansion, so in practice there is just one way to η-expand *E*.

Observe that, in the equation defining `mapSucc`, the argument `xs` occurs once on the left hand side of =, and one as the last argument that is passed to **map**. We can η-contract this definition of `mapSucc` and obtain an equivalent, more compact one:

```
mapSucc :: [Int] → [Int]
mapSucc = map (1 +)
```

Note also that we used a section, instead of defining the anonymous successor function. In a similar way we can η-contract the previous definition of `filterEven`, and obtain:

```
filterEven :: [Int] → [Int]
filterEven = filter even
```

■

13.5 Exercises

Exercise 13.1. Define a function `predecessor` as a specialization of `(+)`.

Exercise 13.2. Define the function **negate** (negation of an integral number) as a specialization of `(-)`.

Exercise 13.3. Define the function **recip** (the reciprocal of a fractional number) as specialization of a predefined operator.

Exercise 13.4. Determine which of the following expressions are well typed and, for those that are well typed, determine their most general type. Check your answers with GHCi:

1. `(+) 1 2 3`
2. `(-) 1`
3. `(**) (if True then sin else cos)`
4. `(**) (if True then 1 else 2)`
5. `(+) fromIntegral 1`
6. `(+) (fromIntegral 1)`
7. `1 (+) 2`
8. `if True then (+) else (/) 1`
9. `(if True then (+) else (/)) 1`
10. `if (&&) True True then (++) else (\s → \t → s ++ ","++ t)`

Exercise 13.5. Let **curry** and **uncurry** be the functions defined in Section 13.3 (note that these functions are predefined in the standard Haskell library). Determine the most general type and the behavior of the following functions:

1. `curry (uncurry divides . \(x, y) → (y, x))`
2. `curry (uncurry (/) . \(x, y) → (y, x))`
3. `curry fst`
4. `uncurry (*) . (\x → (x, x))`
5. `(==) 0 . uncurry mod . (\x → (x, 2))`

Part V

User-defined data types

Chapter 14

Type aliases and enumerations

Besides the predefined primitive and structured data types, Haskell gives the programmer the ability to *define types*, either as aliases for existing types, or as completely new types. In the latter case, the programmer must specify precisely which values inhabit the new type. In this chapter we introduce type aliases and we begin our exploration of Haskell algebraic data types with the simplest form a new data type definition, the *enumeration*.

14.1 Type aliases

Very often we use an existing Haskell type to represent values that, logically speaking, have a clear meaning. For example, it is often the case that Haskell strings can be used to represent names of some sort. In these cases, it would be better, for readability and documentation purposes, to be able to use a type with a more informative name, which however is completely equivalent to the original type. We can do so by defining *type aliases* in Haskell scripts:

```
type Name = String
```

The above definition states that `Name` is an alias for `String`. The two type names can be used interchangeably: declaring that a value has type `Name` is the same as declaring that the same value has type `String`. However, `Name` gives a more precise idea of what some value is meant to represent.

In Section 9.6 we have defined a little library of functions to work with association lists. An association list is nothing but a list of pairs. For example, the function `lookup` has type `Eq a ⇒ a → [(a, b)] → b`, where the first argument, of type '`a`', is the key for which we are seeking an association, the second argument, of type `[(a, b)]`, is the association list, and the result, of type '`b`', is the value associated with the key. It is a pity, in this context, that the type of `lookup` does not say explicitly that its first argument must be an association list: `[(a, b)]` is the generict type of lists of pairs, but `lookup` really make sense only for association lists. In such cases, it could be convenient to define a type alias that is structurally equivalent to `[(a, b)]` but whose name provides an explicit hint as to what the values of that type are meant to represent. Unlike the example of `Name` above, however, we must take into account that the type `[(a, b)]` depends on the type variables `a` and `b`. In this case we can define a *polymorphic type alias* thus:

```
type Map a b = [(a, b)]
```

and now we can define the functions in Section 9.6 with the following types:

```
lookup :: (Eq a) ⇒ a → Map a b → b
insert :: a → b → Map a b → Map a b
delete :: (Eq a) ⇒ a → Map a b → Map a b
member :: (Eq a) ⇒ a → Map a b → Bool
```

Observe that the use of the type alias `Map` does not prevent us from mistakently applying one of the functions above to a list of pairs that does not represent an associative list. This is because the types `Map a b` and `[(a, b)]` are *structurally equivalent*, as by the definition of `Map`: any value of type `Map a b` can be used where a value of type `[(a, b)]` is expected, and vice versa.

Because of the structural equivalence between a type alias and its definition, type aliases can be useful for abbreviating long, complex types, especially if these occur several times in a Haskell script.

14.2 Enumerations

When defining a *new data type*, we need to provide at least two pieces of information: the *name* of the new type, and the *values* of the new type. The simplest form of new type definition is thus as simply as:

data T = C_1 | C_2 | ··· | C_n

where T, which is an identifier beginning with a capital letter, is the name of the new type being defined, and C_1, C_2, ..., C_n, which are identifiers beginning with a capital letter and are called *constructors*, are all and the only values of the type T. A data type like T is sometimes called *enumeration* because its definition explicitly enumerates all the values of type T. Going back to the intuition of Chapter 3, where we have seen that we can think of types as of sets of values, the type definition for T corresponds to the definition

$T = \{ C_1, C_2, \ldots, C_n \}$

As an example, we can define a Haskell type `Dir` representing the four cardinal directions thus:

```
data Dir = North | South | East | West
```

With this definition in place, the identifiers `North`, `South`, `East`, and `West` are values of type `Dir`:

```
> :type North
North :: Dir
> :type South
South :: Dir
> :type West
West :: Dir
> :type East
East :: Dir
```

however if we try to print one of them we get an error message:

```
> North
<interactive>:1:0:
    No instance for (Show Dir)
      arising from a use of ‘print’ at <interactive>:1:0-4
    Possible fix: add an instance declaration for (Show Dir)
    In a stmt of a ’do’ expression: print it
```

because GHCi does not know how to show a value of type Dir. For simple types like Dir Haskell can define reasonable default code to show values of such types. It suffices to add a suitable declaration when Dir is defined, saying that we also want Dir to be an instance of the class **Show**. While we are at it we make Dir also an instance of the class **Eq**, which will allow us to use equality and inequality operators:

```
data Dir = North | South | East | West
           deriving (Eq, Show)
```

We are now able to print values of type Dir and to compare them:

```
> North
North
> North == North
True
> North == South
False
```

More generally, every value of type Dir is equal only to itself, and it is different from any other value of type Dir. Furthermore, the type Dir is different from any other Haskell type, hence it is not possible to compare values of type Dir with values of any other type:

```
> North == True
<interactive>:1:9:
    Couldn't match expected type ‘Dir’ against inferred type ‘Bool’
    In the second argument of ‘(==)’, namely ‘True’
    In the expression: North == True
    In the definition of ‘it’: it = North == True
> North == 1
<interactive>:1:9:
    No instance for (Num Dir)
      arising from the literal ‘1’ at <interactive>:1:9
    Possible fix: add an instance declaration for (Num Dir)
    In the second argument of ‘(==)’, namely ‘1’
    In the expression: North == 1
    In the definition of ‘it’: it = North == 1
```

The last error is particularly interesting: GHCi does not say that North and 1 cannot be compared because they are different. Rather, it suggests that in order for North and 1 to be comparable, the type Dir should be made an instance of the class **Num**.

Since North, South, East, and West are the only values of type Dir and they are all different from each other, they are in fact the *canonical constructors* of values of type Dir. The most important implication of this fact is that they can be used in patterns for matching values of type Dir. For example, we can define a function left like this:

```
left :: Dir → Dir
left North = West
left West = South
left South = East
left East = North
```

and consequently a function turnAround as follows:

```
turnAround :: Dir → Dir
turnAround = left . left
```

Needless to say, values of type Dir can be used in every context where a value can be used, provided that the usual type constraints are satisfied.

Example 14.1. We now realize that **Bool** is nothing but an enumeration with two constructors **False** and **True** whose definition is the following:[1]

```
data Bool = False | True
            deriving (Eq, Ord, Show)
```

In fact, any basic Haskell with a finite number of values, such as **Int**, **Char**, and **Float**, can be seen as an enumeration with a special syntax for denoting its constructors. ■

Enumeration are particularly useful for representing data types consisting of a fixed, relatively small number of possible values. In Section 9.6 we have seen how to represent week days as either string or numbers, and how to give a mapping between these two representations. Neither representation is completely satisfactory though. Representing week days as numbers is efficient (in terms of memory occupation and of computational complexity for comparing week days), but numbers are not informative enough. An occurrence of 1 could either stand for the number one or for the first working day. Both interpretations are possible, and only by looking at the context in which the literal occurs it is possible to understand which is the correct one. Furthermore, GHCi cannot detect whether we are using a number that is not in the range 1-7 as a week day. This may introduce subtle errors in the code that are hard to spot. Representing week days as strings has the advantage that the program becomes more informative, since week days are spelled out explicitly in the code. However, manipulating strings is generally less efficient than manipulating numbers (recall that Haskell strings are lists of characters). Furthermore, it is again possible to inadvertently use nonsensical strings in places where a week day is expected. In this case an enumeration like the following is a clearly winning choice:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday
         | Saturday | Sunday
           deriving Ord
```

Enumerations provide a compact, informative, and efficient way of representing values from a small (finite) set. Furthermore, the type signature of functions using enumerations must report the name of the enumeration, thus improving the code readability and documentation. For example, the type of a hypothetical function tomorrow would clearly state that it transforms values of type Day:

```
  tomorrow :: Day → Day
```

14.3 Exercises

Exercise 14.1. Implement the function tomorrow whose type has been shown in Section 14.2.

Exercise 14.2. Implement a function yesterday that is the inverse of tomorrow.

Exercise 14.3. Devise a non-recursive function after that, when applied to a Day d and a non-negative integer number n, computes the day of the week corresponding to n days after d.

Exercise 14.4. Define an enumeration QuasiBool whose constructors are No, Yes e Unknown where No and Yes represent "false" and "true" respectively, and Unknown represents an uncertain value that can be either "false" or "true". Then define:

[1]Actually the **Bool** type definition derives even more instances of type classes that are not treated in this tutorial.

1. binary functions `qand` and `qor`, whose behavior must be analogous to the operators `&&` and `||`, but that work with `QuasiBool` values;

2. the unary function `qnot`, analogous to **`not`**;

3. the generalizations of `qand` and `qor` over lists of `QuasiBool` values as specializations of existing functions in the Haskell library;

4. conversion functions from `QuasiBool` to **`Bool`** values and from **`Bool`** to `QuasiBool` values.

In defining the functions above, make sure that the returned value is the most precise, where both `No` and `Yes` are more precise than `Unknown`.

Chapter 15

Compound data types

In this chapter we extend algebraic data types by allowing constructors to have parameters. This permits the programmer to define new types in terms of existing ones. Also, just like type aliases, algebraic data types can have type parameters.

15.1 Constructors with parameters and injectors

Sometimes it is convenient to define a type alias for an existing Haskell type for representing some specific data, since doing so allows us to inherit all of the operators and functions associated with that type. The problem of this approach, which we have highlighted in Section 14.1, is that we might accidentally use a value of the "new" type in place of a value of the original type or, maybe worse, we can do the opposite. Since the two types are structurally equal, the Haskell compiler cannot provide any active support to help the programmer in finding these errors.

One way to solve the problem is to define a *new* type that is *isomorphic* to an existing one. This can be achieved by defining a type with a parametric constructor. For example:

```
data Age = Age Int
           deriving (Eq, Ord, Show)
```

defines a new type `Age` (as usual, the name of the type is the identifier preceeding `=`) with one constructor `Age` (the identifier following `=`) that has one parameter of type **`Int`**. The values of type `Age` have the form `Age` n where n has type **`Int`**. Following the set-theoretic interpretation of types, we can write

`Age = { Age` n `|` $n \in$ **`Int`** `}`

For example:

```
> Age 1
Age 1
> Age 32
Age 32
> :type Age 1
Age 1 :: Age
> :type Age 32
Age 32 :: Age
```

We can think of a value `Age` n as value of type **`Int`** to which we have attached a label `Age`, that distinguishes it from any other value of type **`Int`**. It is easy to see that the type

`Age` and the type **`Int`** are isomorphic, namely there is a value of type `Age` for every value of type **`Int`**, and vice versa, and yet they are incompatible types:

```
> Age 1 == 1
<interactive>:1:9:
    No instance for (Num Age)
      arising from the literal ‘1’ at <interactive>:1:9
    Possible fix: add an instance declaration for (Num Age)
    In the second argument of ‘(==)’, namely ‘1’
    In the expression: Age 1 == 1
    In the definition of ‘it’: it = Age 1 == 1
```

In this particular example, the type name `Age` coincides with the name of its constructor, but this is not necessary. In any case, no confusion between the two names can arise, as Haskell is always able to understand whether an occurrence of the identifier `Age` stands for a type name or for a constructor name from the context in which the identifier occurs.

Another way of looking at the constructor `Age` is as an *injector* that takes a value of type **`Int`** and turns it into a value of type `Age`. This intuition is strengthened by the observation that the constructor `Age` is in fact a Haskell function whose domain and codomain are just the types **`Int`** and `Age` respectively:

```
> :type Age
Age :: Int → Age
```

Haskell does not provide the inverse function, but allows us to use the constructor `Age` in patterns. This way, we can extract the value n from a value of the form `Age` n:

```
ageToInt :: Age → Int
ageToInt (Age n) = n
```

The parentheses around the pattern `Age n` are necessary for otherwise Haskell would interpret `Age` and `n` as two distinct patterns.

Since `Age` and **`Int`** are different types, we cannot mistakenly confuse one for the other. If we need to convert an integer into an age, we must use the injector `Age`. If we need to go the other way around, we have to match the value of type `Age` against a pattern, or we have to resort to the function `ageToInt` above. In any case, the point where a value of one type is used in place of a different type is now flagged by some explicit code, which makes the program more robust and self-explaining.

In order to compare ages, it could be possible to write appropriate functions such as the followings:

```
ageLt :: Age → Age → Bool
ageLt (Age m) (Age n) = m < n

ageMax :: Age → Age → Age
ageMax (Age m) (Age n) = Age (max m n)
```

This quickly becomes tedious since all these functions are doing is to extract the interger values from a values of type `Age`, compare the integer values, and possibly injecting back the result into the type `Age`. Haskell is able to generate these functions automatically if asked to do so, which is achieved by specifing the classes **`Eq`**, **`Ord`**, and **`Show`** in the definition of `Age` above. Thus it is legal to write:

```
> Age 1 < Age 32
True
> max (Age 1) (Age 32)
```

```
Age 32
```

In Section 8.4 we have seen how to represent rational numbers as pairs of integer numbers. This representation suffers from the same problems that we have mentioned above, in that a pair of integer numbers can be mistakenly used where a fraction is expected, or vice versa. We could solve the problem by defining a data type like the following one:

```
data Fraction = Fraction (Int, Int)
```

Alternatively, we can define the constructor `Fraction` as having *two* parameters of type **`Int`**, as opposed to one parameter of type `(`**`Int`**`, `**`Int`**`)`. This is achieved with the following definition:

```
data Fraction = Fraction Int Int
                deriving (Eq, Ord, Show)
```

The set-theoretic intuition of data types tells us that `Fraction` is the type

$$\texttt{Fraction} = \{ \texttt{Fraction}\ m\ n \mid m \in \texttt{Int},\ n \in \texttt{Int} \}$$

that is, `Fraction` is isomorphic to the cartesian product **`Int`** $\times$ **`Int`** or, equivalently, to values of type `(`**`Int`**`, `**`Int`**`)`. Values of type `Fraction` have the form `Fraction` m n for two arbitrary integer numbers m and n. For example, we have:

```
> Fraction 1 2
Fraction 1 2
> :type Fraction 1 2
Fraction 1 2 :: Fraction
```

which represents the number $\frac{1}{2}$. As before, the constructor `Fraction` is an injector, except that it now takes two integer numbers to produce a value of type `Fraction`. The type of `Fraction` shows that it is a curried binary function:

```
> :type Fraction
Fraction :: Int → Int → Fraction
```

Because of the **`deriving`** clause in the definition of the type `Fraction`, we can use the standard Haskell relational operators for comparing fractions:

```
> Fraction 1 2 == Fraction 1 2
True
> Fraction 1 2 == Fraction 1 3
False
```

but the default ordering of values of type `Fraction` is the same used for pairs of type `(`**`Int`**`, `**`Int`**`)`, and we have already seen that this may cause problems:

```
> Fraction 1 2 < Fraction 1 3
True
```

However, there is an advantage over the representation of fractions as simply pairs of integers: `Fraction` is a new type that is different from `(`**`Int`**`, `**`Int`**`)`. This means that it is possible to provide an implementation of the instance **`Ord`** `Fraction` that matches the intended semantics of values of type `Fraction`. The interested reader should refer to one of the several books on Haskell programming for details about defining new type class instances.

15.2 Compound data types

In Section 14.2 we have seen how to define a new data type with several constructors. In Section 15.1 we have seen how to define a new data type with only one (parameterized) constructor. The two forms of definition can be freely mixed so as to define new data types with several constructors, each having zero, one, or more parameters independently of the other constructors. This can be useful for defining the "union" of two or more existing data types. For example, we can define a single type `Number` that stands for either interger or floating-point numbers thus:

```
data Number = I Int | F Float
              deriving (Eq, Ord, Show)
```

Here we have defined a new type `Number` that has two constructors, `I` and `F`. The constructor `I` has one parameter of type **`Int`**, and the constructor `F` has one parameter of type **`Float`**. Corresponding to the two constructors, there are two canonical ways of creating a value of type `Number`:

```
> I 10
I 10
> F 3.5
F 3.5
> :type I 10
I 10 :: Number
> :type F 3.5
F 3.5 :: Number
```

Observe that both `I 10` and `F 3.5` have the same type. Intuitively, `Number` is the union of the types **`Int`** and **`Float`**, and `I` and `F` allow us to inject values of these types into `Number`:

```
> :type I
I :: Int → Number
> :type F
F :: Float → Number
```

The union is *disjoint*, in the sense that each value of type **`Int`** is labeled by the `I` constructor and each value of type **`Float`** is labeled by the `F` constructor. That is, the set-theoretic interpretation of `Number` is:

$$\texttt{Number} = \{\ \texttt{I}\ m \mid m \in \textbf{Int}\ \} \cup \{\ \texttt{F}\ f \mid f \in \textbf{Float}\ \}$$

We can now define functions that work with heterogeneous numbers, such as `addNumber` and `divNumber` below:

```
addNumber :: Number → Number → Number
addNumber (I m) (I n) = I (m + n)
addNumber (I m) (F f) = F (fromIntegral m + f)
addNumber (F f) (I m) = F (f + fromIntegral m)
addNumber (F f) (F g) = F (f + g)

divNumber :: Number → Number → Number
divNumber (I m) (I n) | n ‘divides‘ m = I (m ‘div‘ n)
divNumber (I m) (I n) = F (fromIntegral m / fromIntegral n)
divNumber (I m) (F f) = F (fromIntegral m / f)
divNumber (F f) (I m) = F (f / fromIntegral m)
divNumber (F f) (F g) = F (f / g)
```

Because of the **deriving** clause in the definition of `Number`, default relational operators are available for values of type `Number`, but once again they don't match the expected semantics. In particular, two values of type `Number` are equal only if they have been built using the same constructor, and if the corresponding parameters are equal. Moreover, the default ordering is such that every value of the form `I` m is smaller than any value of the form `F` f, regardless of m and f. Hence we have:

```
> I 1 == F 1
False
> I 10 < F 1
True
```

Once again, the solution is to explicitly provide an implementation of the instances **Eq** `Number` and **Ord** `Number`.

Example 15.1 (values with errors). There are several ways of implementing *partial functions*, namely functions that are not defined for some values of their domain. One is to let the function terminate abnormally the whole program, if applied to an argument that it cannot handle. Another possibility is to reserve one or more values from the codomain to indicate that an error has occurred and that the function was unable to properly yield a result. Compound data types give us yet another option: that of defining a type with one parameterized constructor that indicates the successful termination of the function. The parameter carries the value returned by the function. Then, we have one (or more) constructors indicating that an error (and possibly which error) has occurred.

Here we define a type `MaybeInt` and a safe version of the integer division that returns `NO` if a division by zero is attempted:

```
data MaybeInt = OK Int | NO
                deriving (Eq, Show)

safeDiv :: Int → Int → MaybeInt
safeDiv _ n | n == 0 = NO
safeDiv m n = OK (m ‘div‘ n)
```

After applying `safeDiv`, the programmer has to explicitly match the resulting value to see whether it is of the form `OK` n for some integer value n or of the form `NO`:

```
> safeDiv 5 3
OK 1
> safeDiv 5 0
NO
```

This way, every place where a potential division by zero can occur requires an explicit handling from the programmer. ■

15.3 Polymorphic compound data types

In Example 15.1 we have defined a type `MaybeInt` that allow us to represent "optional values of type **Int**". A value of the form `OK` n means that we do have an integer value (n), whereas a value of the form `NO` means that we have no integer value. It should be evident that in different contexts one might need other types for representing optional floating-poing number, optional booleans, optional characters, optional strings and so on and so forth. All of these types would essentially have the same structure, with one parameterized constructor with a parameter of the type that we want to make optional, and one constant constructor. This clearly calls for a mechanism to define a new type that is parametric in another type (possibly multiple types). In Haskell this is achieved by defining *polymorphic data types* such as the following:[1]

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

Here we have defined a type called **Maybe** which is parametric in another type represented by the type variable a. The constructor **Just** has one parameter of type ‘a’, whereas the constructor **Nothing** is constant. Then, the type MaybeInt is isomorphic to **Maybe Int**, the only difference being in the names of the constructors. We can now rewrite safeDiv using **Maybe** as follows:

```haskell
safeDiv :: Int → Int → Maybe Int
safeDiv _ n | n == 0 = Nothing
safeDiv m n = Just (m ‘div‘ n)
```

Being **Maybe** a polymorphic data type, it is no surprise that its constructors are polymorphic:

```
> :type Just
Just :: a → Maybe a
> :type Nothing
Nothing :: Maybe a
```

The **Maybe** type is used in the codomain of several partial functions in the Haskell standard library.

Example 15.2 (safe lookup). The function **lookup** that we have defined in Section 14.1 is implemented in the Haskell standard library as follows:

```haskell
lookup :: (Eq a) ⇒ a → [(a, b)] → Maybe a
lookup _ [] = Nothing
lookup k ((k’, x) : _) | k == k’ = Just x
lookup k (_ : xs) = lookup k xs
```

Observe that the codomain of the function is now **Maybe** a and not simply ‘a’ as it were in our previous implementation. The function has two base cases: when the associative list is empty the key is not present in the list, so the result is **Nothing**; when the head of the associative list is a pair whose first component is equal to the key, the associated value is returned (after being injected into the **Maybe** a type by means of the **Just** constructor). ■

Example 15.3. The **Maybe** type permits the representation of optional values, but the constructor **Nothing** does not carry any information that justifies why the value is missing. Such information can be an error code, a value that caused the error, a string containing an informative message, and so on. The type **Either** generalizes **Maybe** by defining two constructors **Left** and **Right**, each one having a parameter. The **Right** constructor is meant to represent a “right” result, whereas the **Left** constructor is meant to represent an error or otherwise unexpected result. The type **Either** is defined thus:

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

Observe that **Either** has two type parameters, ‘a’ and ‘b’, representing the type of the parameter of the **Left** and **Right** constructors respectively. Consequently the two constructors have the following types:

[1]Since the type **Maybe** is generally useful, it is already defined in the Haskell standard library.

```
> :type Left
Left :: a → Either a b
> :type Right
Right :: b → Either a b
```

More generally, the **Either** a b type represents the disjoint union of two types a and b. For example, **Either Int Float** is isomorphic to the type Number introduced in Section 15.2. ■

15.4 Pattern matching without application

In Example 9.8 we have defined a function **zip** for combining two lists of elements so as to produce a third list whose elements are pairs of corresponding elements in the original lists. If the two lists being combined have different lengths, the longest list is truncated. An alternative possibility is to return a result only if the two lists have the same length, which makes a case for using the **Maybe** type. The idea is to write a function combine that returns an optional list containing the combination of the two original lists. The first three equations for combine are fairly easy:

```
combine :: [a] → [b] → Maybe [(a, b)]
combine [] [] = Just []
combine [] _ = Nothing
combine _ [] = Nothing
```

The missing equation should deal with the case when both lists are nonempty, in particular the first has the form (x : xs) and the second has the form (y : ys). Intuitively, we should recursively apply combine to xs and ys, and then we should cons the pair (x, y) in front of the resulting list. However, we must also take into account the fact that the recursive application of combine returns an optional list, not a list. More precisely, if the recursive application returns **Nothing**, then this means that the tails xs and ys do not have the same length, and the overall result should be **Nothing**. If the recursive application returns **Just** l for some list l, then the overall result should be **Just** ((x, y) : l). In practice, we need to perform a pattern match on the value resulting from the recursive application of combine, while up to now we have always performed pattern matching when applying a function to its arguments. Haskell provides another construct for inspecting a value by pattern matching at any point in an expression and the last equation of combine can be implemented thus:

```
combine (x : xs) (y : ys) =
    case combine xs ys of
      Nothing → Nothing
      Just l → Just $ (x, y) : l
```

The right hand side of the equation is an expression beginning with the keyword **case**. This defines a pattern matching of the expression that follows **case**, the recursive application of combine in this example. Following the keyword **of** are a list of aligned *rules*: each rule begins with a pattern followed by → and by an expression. The value of the expression after **case** is matched against the patterns in the rules. When a pattern is found that matches the value, the corresponding expression is evaluated. In case of overlapping patterns, the first-match policy applies.

```
> combine [1, 2, 3] ['a', 'b', 'c']
```

```
Just [(1,'a'),(2,'b'),(3,'c')]
> combine [1, 2, 3] ['a', 'b']
Nothing
```

Unlike pattern matching in function definitions, where it is possible to define patterns for several arguments, the **case**-**of** construct allows to match just one value against a set of patterns. In case more values must be matched simultaneously, it is possible to build up a tuple and do pattern matching on it.

When p is an irrefutable pattern, the use of an expression of the form

```
case E of
  p → E′
```

seems unnecessarily heavyweight. After all, we know that the value resulting from E is matched by p and, most likely, the only thing we need is to give names to the components of this value and use them in the continuation E'. In such cases, Haskell provides a lighter notation:

```
let p = E in E′
```

which has exactly the same effect as the **case**-**of** above. In fact, the **let**-**in** constructs we have used so far are just a special case of this one, where the pattern was always a single name, namely an irrefutable pattern! We have already seen an example of this style of local definition at the end of Section 8.5.

Example 15.4. Suppose we want to define the inverse function of **zip**, namely a function **unzip** that takes a list of pairs a produces a pair of lists obtained by grouping all the first and second components from the list of pairs. We can define **unzip** as follows:

```
unzip :: [(a, b)] → ([a], [b])
unzip [] = ([], [])
unzip ((x, y) : ps) = (x : xs, y : ys)
  where
    (xs, ys) = unzip ps
```

Unlike **zip**, the function **unzip** is always defined. The recursive application produces a pair of lists from which the two components `xs` and `ys` must be extracted. Instead of using **fst** and **snd** for this purpose, we match the result of the recursive application against the irrefutable pattern `(xs, ys)`, which allows us to directly give a name to the two compoe-nents. ■

15.5 Exercises

Exercise 15.1. Define an algebraic data type for representing complex numbers, composed by a real and an imaginary part. Define functions for:

1. adding, subtracting, multiplying, and dividing complex numbers;

2. compute the modulus, extract the part, extract the imaginary part of a complex number.

Exercise 15.2. Define a function `maybeMap :: (a → Maybe b) → [a] → [b]` that behaves like **map**, except that the transformation function returns an optional value and the resulting list contains only those different from **Nothing**, stripped off the constructor **Just**.

Exercise 15.3. Define a function `stripJust :: [Maybe a] → [a]` that removes `Nothing` values from a list and strips the constructor `Just` off the remaining ones.

Exercise 15.4. Define `maybeMap` (Exercise 15.2) as a combination of `map` and `stripJust` (Exercise 15.3).

Exercise 15.5. Define a function `splitEither :: [Either a b] → ([a], [b])` that separates the parameters of the constructors `Left` and `Right` of the original list into a pair of lists.

Exercise 15.6. Define two functions `showNumber :: Number → String` and `readNumber :: String → Numb` for convering values of type `Number` to strings and strings to values of type `Number` respectively.

Exercise 15.7. Define a type alias `Range = (Maybe Int, Maybe Int)`, where a `Nothing` component denotes $-\infty$ (if it is the first component) or $+\infty$ (if it is the second component). Define functions for computing the intersection and the union of two ranges (the union of two ranges is the smallest range that includes both).

Chapter 16

Recursive data types

All of the data types introduced so far can have parameterized constructors where the type of the parameters are existing data types, possibly predefined by Haskell. In order to define recursive data types such as lists and trees we relax this condition and permit parameters of a constructor to have the very same type that the constructor belongs to. A data type where at least one constructor mentions the same data type in one of its parameters is said to be a recursive data type.

16.1 Inductive interpretation of recursive types

Consider the data type

```
data Nat = Z | S Nat
           deriving (Eq, Ord, Show)
```

which comprises two constructors: the constant constructor `Z` and the constructor `Z` which has a parameter of type `Nat`. Unlike previous data types, it is not obvious what the set-theoretic interpretation of `Nat` is, since it looks like `Nat` is defined in terms of itself:

```
  Nat = { Z } ∪ { S n | n ∈ Nat }
```

One way to formalize this definition is to say that `Nat` is the *smallest set* such that

- `Z` is in the set-theoretic interpretation of `Nat`;
- if n is in the set-theoretic interpretation of `Nat`, then so is `S` n.

The inductive definition of the type `Nat` is matched by the types of its constructors:

```
> :type Z
Z :: Nat
> :type S
S :: Nat → Nat
```

The type `Nat` permits the representation of natural numbers in terms of two canonical constructors: `Z` stands for “zero” and `S` stands for “successor”. The definition of `Nat` says that every natural number is zero, or is the successor of some other natural number. Under this interpretation, it is straightforward to provide functions that manipulate natural numbers. The function that checks whether two natural numbers are equal is a simple as:

```
eqNat :: Nat → Nat → Bool
eqNat Z Z = True
eqNat (S m) (S n) = eqNat m n
eqNat _ _ = False
```

The sum of two natural numbers m and n must be defined recursively: if m is zero, then the sum of m and n is n. If m is the successor of some other natural number m', then the sum of m and n is the successor of the sum of m' and n. The recursion is well founded since we have a base case and, at each recursive step, the first natural number becomes smaller. The implementation of this operation is just a matter of rephrasing the text into Haskell code:

```
plusNat :: Nat → Nat → Nat
plusNat Z n = n
plusNat (S m) n = S (plusNat m n)
```

The multiplication of two natural numbers can be implemented similarly, by using `plusNat` as an auxiliary function:

```
mulNat :: Nat → Nat → Nat
mulNat Z _ = Z
mulNat (S m) n = plusNat n (mulNat m n)
```

With these definitions we are building basic arithmetics without the help of any native data type:[1]

```
> plusNat (S (S Z)) (S (S (S Z)))
S (S (S (S (S Z))))
> mulNat (S (S Z)) (S (S (S Z)))
S (S (S (S (S (S Z)))))
```

The definition of `Nat` generalizes to that of lists. For example, the data type for representing arbitrarily long lists of integer numbers can be defined thus:

```
data ListInt = Nil | Cons Int ListInt
```

where `Nil` represents the empty list and `Cons` n l is a list with head n and tail l. We can further generalize this data type by abstracting over the type of elements of the list, and obtain:

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

Observe that `List` and the predefined list type in Haskell are isomorphic, the only difference being that Haskell provides the programmer with some syntactic sugar so that one writes `[1, 2, 3]` or `1 : 2 : 3 : []` as opposed to the more verbose `Cons 1 (Cons 2 (Cons 3 Nil))`. A function such as `length` for the type `List` would be defined thus:

```
length :: List a → Int
length Nil = 0
length (Cons _ l) = 1 + length l
```

As an exercise, the reader may want to re-write a few more functions that work with lists represented with `Nil` and `Cons`.

[1]Beware that working with natural numbers represented in this way is extremely inefficient!

16.2 Case study: binary search trees

Recursive data types allow us to easily define any data structure that can be expressed inductively. As an example, the type `BTree a` of binary trees with labelled-nodes where the label has type ‘a’ can be inductively described thus:

- `Empty` is a binary tree of type `BTree a`;
- if n is a value of type a and l and r are binary trees of type `BTree a`, then `Node` $n\ l\ r$ is a binary tree of type `BTree a`.

The definition of `BTree` follows naturally:

```
data BTree a = Empty | Node a (BTree a) (BTree a)
               deriving Show
```

We can compute the depth of a binary tree as a simple recursive function that navigates the tree top-down:

```
depth :: BTree a → Int
depth Empty = 0
depth (Node _ l r) = max (depth l) (depth r)
```

Similarly, we can collect all the labels of the tree thus:

```
labels :: BTree a → [a]
labels Empty = []
labels (Node n l r) = labels l ++ [n] ++ labels r
```

These definitions suggest that we can provide higher-order functions to manipulate binary trees just as we did for lists. In these examples, we are folding the tree by providing a value to be used when the tree is empty (`0` or `[]`), and by using a specific function (**`max`** or ++) for combining the foldings of the subtrees of a nonempty subtree with its label. By abstracting over the value and the combining functions, we realize that both `depth` and `labels` have the same structure, as by the following function `fold`:

```
fold :: a → (b → a → a → a) → BTree b → a
fold x _ Empty = x
fold x f (Node n l r) = f n (fold x f l) (fold x f r)
```

Then `depth` and `labels` are simple specializations of `fold`:

```
depth :: Btree a → Int
depth = fold 0 (\_ → max)

labels :: Btree a → [a]
labels = fold [] (\n l r → l ++ [n] ++ r)
```

A *binary search tree* is a binary tree such that every node of the form `Node` $n\ l\ r$ has the property that n is greater than every label found in l and smaller than every label found in r. We devote the rest of this section to implementing a handful of functions for manipulating binary search trees, starting from the function for checking whether a label occurs in a binary search tree:

```
lookup :: (Ord a) ⇒ a → BTree a → Bool
lookup _ Empty = False
lookup x (Node y l r) | x == y = True
                      | x < y = lookup x l
                      | otherwise = lookup x r
```

The function for retrieving the least label in a nonempty binary search tree can be implemented thus:

```
least :: BTree a → a
least (Node x Empty _) = x
least (Node _ l _) = least l
```

Observe that the `least` function is partial, it is not defined on empty binary trees, and that it does not mandate the type a to be an instance of the type class **Ord**. This is because technically no method of **Ord** is used inside `least`, although the function assumes that the input tree is a binary search tree and thus the least element is found by navigating through the leftmost spine of the tree.

Insertion in a binary search tree is structurally similar to **lookup**, by navigating the tree according to the label being inserted. The insertion is not performed *in-place*. Rather, a new binary search tree is returned where the new label has been inserted at the right place:

```
insert :: (Ord a) ⇒ a → BTree a → BTree a
insert x Empty = Node x Empty Empty
insert x t@(Node y l r) | x == y = t
                        | x < y = Node y (insert x l) r
                        | otherwise = Node y l (insert x r)
```

Deletion of a label from a binary search tree is a little more tricky, because it may require to merge two nonempty subtrees into a binary search tree. If this happens, the least element is deleted from the right subtree and promoted as root of the tree:

```
delete :: (Ord a) ⇒ a → BTree a → BTree a
delete _ Empty = Empty
delete x (Node y Empty r) | x == y = r
delete x (Node y l Empty) | x == y = l
delete x (Node y l r)     | x < y = Node y (delete x l) r
                          | x > y = Node y l (delete x r)
delete x (Node y l r)     | x == y = Node z l r'
    where
      z = least r
      r' = delete z r
```

16.3 Case study: abstract syntax trees

In this section we develop an evaluator for a simple language of expressions whose syntax is described by the grammar in Table 16.1. There are two syntactic categories: E for integer expressions and B for boolean expressions. An integer expression can be an integer constant n, a variable x, the sum of two integer expressions $E_1 + E_2$, a conditional expression `if` B `then` E_1 `else` E_2 that evaluates to E_1 if B is true and to E_2 if B is false, or a local definition `let` $x = E_1$ `in` E_2 that evaluates E_2 where x is bound to the value of E_1.

The Haskell evaluator for this language must adequately represent any *expression* that can be written in the grammar above and we make the simplifying assumption of working directly with *abstract syntax trees* corresponding to terms that can be built from the categories E and B. Haskell data types are natural candidates for representing abstract syntax trees, since they allow the representation of labeled trees where the label of each nodes identifies one of the productions in the grammar in Table 16.1. We notice that, in the

Table 16.1: Syntax of a simple language of expressions.

E	::=		**expression**
		n	(constant)
	|	x	(variable)
	|	$E + E$	(sum)
	|	`if` B `then` E `else` E	(conditional)
	|	`let` $x = E$ `in` E	(binding)
B	::=		**condition**
		b	(constant)
	|	$B \wedge B$	(conjunction)
	|	$\neg B$	(negation)
	|	$E = E$	(equality)

grammar, the productions for the categories E and B may refer to the corresponding category. For example, we have $E \to E + E$ for integer expressions and $B \to B \wedge B$ for boolean expressions. This suggests that the data types we will define are going to be recursive. Additionally, we observe that the productions for E refer to the B category, and that the productions for B refer to the E category. That is, the categories E and B are mutually dependend. This makes a perfect case for representing them by means of mutually recursive data types.

We define two data types `IExpr` and `BExpr` representing the categories E and B respectively:

```
type Name = String

data IExpr = IConst Int
           | Var Name
           | Add IExpr IExpr
           | If BExpr IExpr IExpr
           | Let Name IExpr IExpr
           deriving Show

data BExpr = BConst Bool
           | And BExpr BExpr
           | Not BExpr
           | Eq IExpr IExpr
           deriving Show
```

Observe that we have one constructor for each production in the grammar, and that the parameters of the constructors reflect the relevant information in the body of the production. For example, the `If` consturctor has three parameters corresponding to the test of the conditional expression, which must be a value of type `BExpr`, and the expressions corresponding to both branches of the conditional expression, both of type `IExpr`. For readability purposes, we have defined a type alias `Name` that is equivalent to **`String`**.

With this data types in place, we can represent the expression

$$\texttt{let}\ x = 32\ \texttt{in}\ \texttt{if}\ \neg(x = 32)\ \texttt{then}\ x\ \texttt{else}\ (x + 1)$$

as the value

```
example :: IExpr
example = Let "x" (IConst 32)
                  (If (Not (Eq (Var "x") (IConst 32)))
                      (Var "x")
                      (Add (Var "x") (IConst 1)))
```

The Haskell evaluator needs an environment to keep track of the bindings between names and values. We represent the binding as an association list whose keys are of type `Name` and whose values are of type **`Int`**:

```
type Env = [(Name, Int)]
```

As the `IExpr` and `BExpr` data types are mutually recursive, so must be the two evaluator functions that we are going to define:

```
evalI :: Env → IExpr → Int
evalI _ (IConst n) = n
evalI env (Var x) =
  case lookup x env of
    Nothing → error $ "unknown␣variable:␣" ++ x
    Just v → v
evalI env (Add e1 e2) = evalI env e1 + evalI env e2
evalI env (If b e1 e2) | evalB env b = evalI env e1
                       | otherwise = evalI env e2
evalI env (Let x e1 e2) = evalI env' e2
    where
      env' = (x, evalI env e1) : env

evalB :: Env → BExpr → Bool
evalB _ (BConst b) = b
evalB env (And b1 b2) = evalB env b1 && evalB env b2
evalB env (Not b) = not $ evalB env b
evalB env (Eq e1 e2) = evalI env e1 == evalI env e2
```

Each function does a pattern matching on the structure of the expression being evaluated and behaves accordingly. In the `Var` case, the variable `x` is looked up in the environment by means of the predefined function **`lookup`**. If `x` is not found in the environment, **`lookup`** returns **`Nothing`** and we terminate the evaluation by printing an error message along with the name of the unknown variable. If `x` is found in the environment, **`lookup`** returns **`Just`** v where v is the value associated with `x`, and we simply return v.

The evaluation of the expression `example` above yields the following result, as expected:

```
> evalI [] example
33
```

whereas the evaluation of an expression contanining undefined variables causes an error:

```
> evalI [] (Var "x")
*** Exception: unknown variable: x
```

16.4 Exercises

Exercise 16.1. Consider the type

```
type BString = End | Zero BString | One BString
```

for representing arbitrarily long strings of bits, where `End` is the empty string, `Zero` s is the string whose least significant bit is 0 and that continues as s, `One` s is the string whose least significant bit is 1 and that continues as s. For example, the value

```
One $ Zero $ One $ One $ End
```

represents the binary number 1101. Define functions for:

1. converting values of type **`Int`** to/from corresponding values of type `BString`;
2. negating a bit string;
3. computing the bitwise conjunction and disjunction of two bit strings;
4. incrementing a bit string by 1 without resorting to the conversion functions above;
5. adding two bit strings without resorting to the conversion functions above and making sure that the complexity of the operation is linear in the size of the two bit strings.

Exercise 16.2. Define a sorting algorithm for lists without duplicate elements based on binary search trees.

Exercise 16.3. Define a function that, when applied to a binary search tree and to a label l, returns the list of labels along the path from the root of tree to l. If l is not in the tree, the function should return the empty list.

Exercise 16.4 (*). Represent sets of values as binary search trees. Define values/functions for:

1. representing the empty set;
2. checking whether a set is empty;
3. create the singleton set $\{v\}$, for every value v;
4. checking whether a value belongs to a set;
5. adding a value to a set;
6. converting a set of values into a list of values;
7. computing the union of two sets;
8. computing the intersection of two sets;
9. computing the difference of two sets.

For items starting from 7 use the function developed for item 6.

Exercise 16.5 (***). Define a function that, when applied to a binary trees, returns **`True`** if the tree is a binary search tree and **`False`** otherwise. Hint: use ranges defined in Exercise 15.7.

Part VI

Programming in the large

Chapter 17

Modules

In this chapter we introduce the module system of Haskell, which has two main purposes: providing a mechanism for decomposing large programs into maneagable, logically coherent modules, and giving the programmer the ability to (partially) conceal the implementation of a module, so as to improve robustness of code that uses the module while not hindering evolution of the module's implementation.

17.1 Organizing large programs

A large Haskell program is usually organized in modules that so that logically related value and type definitions are grouped within the same module. Until now we have always implicitly worked with one module, the one containing all the definitions in the scripts we have defined. This implicitly defined module is called `Main`.

A module is explicitly defined by beginning a Haskell script with a suitable declaration. For example, the script:

```
module A where

x :: Int
x = 1

succ :: Int → Int
succ = (+ 1)
```

defines a module called `A` that contains two value definitions, for `x` and **`succ`**.

Although Haskell does not mandate a specific policy, every Haskell implementation usually draws a connection between the name and content of a module and the name of the physical text files that contains the modules. Typically, a module called `A` will entirely reside in a file named `A.hs`.

When loading the module above into `GHCi`, the prompt reflects the name of the module just loaded:

```
> :load A.hs
[1 of 1] Compiling A                ( A.hs, interpreted )
Ok, modules loaded: A.
*A>
```

The name `A` in the prompt informs that the module `A` is the default scope when searching for names. That is, we can use simply `x` for referring to the name `x` defined in module `A`:

```
> x
1
```

If we try to use the name **succ**, an error occurs:

```
> succ x
<interactive>:1:0:
    Ambiguous occurrence ‘succ’
    It could refer to either ‘A.succ’, defined at A.hs:7:0
                          or ‘Prelude.succ’, imported from Prelude
```

The problem originates because a function called **succ** is also defined in the **Prelude** module that GHCi opens at startup. Thus, referring to simply **succ** is ambiguous. This brings us to a more explicit way of referring to names, by means of *qualifiers*. We can quality **succ** by prepending the name of the module in which it occurs followed by a dot. Thus:

```
> A.succ x
2
```

refers to the name **succ** defined in module A, whereas

```
> Prelude.succ x
2
```

refers to the **succ** name defined in the **Prelude**.

Incidentally the two versions of **succ** we have been using yield the same result when applied to x. However, note that they have different types:

```
> :type A.succ
A.succ :: Int → Int
> :type Prelude.succ
Prelude.succ :: (Enum a) ⇒ a → a
```

In general, names defined in different modules may refer to completely unrelated types or values.

Module names can consist of multiple qualifiers separated by dots. For example, it is perfectly legal to define a module called A.B like the following one:

```
module A.B where

y :: String
y = "A␣module␣with␣longer␣qualifier"
```

After loading this module in GHCi, y can be referred to either by omitting any qualifier (because there is no ambiguity as to which y we are referring to) or by prefixing y with A.B:

```
> y
"A␣module␣with␣longer␣qualifier"
> A.B.y
"A␣module␣with␣longer␣qualifier"
```

The ability of using multiple qualifiers seems to allow the programmer to create a *hierarchy* of modules. In the example above, it looks like the module A.B is embedded within module A. In practice, Haskell does not enforce this hierarchy in any way: modules A and A.B are completely unrelated to each other. Multiple qualifiers can be useful when designing large libraries (such as the Haskell standard library) as a tool for organizing modules.

17.2 Importing definitions

Before referring to names in other modules, these names must be explicitly imported with an **import** statement in the script:

```
import Data.List

least :: (Ord a) ⇒ [a] → a
least = head . sort
```

In its simplest form, an *import* statement is made of the keyword **import** followed by the full name of the module to be imported. Importing a module makes every name defined in the module to be visible in the scope where the **import** statement occurs. In the script above we have imported all the definitions in module Data.List, in particular the **sort** function:

```
> least "hello"
'e'
```

It is *not* an error to import a module and to redefine a name that also occurs in the imported module. For example, the module Data.List defines a **minimum** function that finds the least element of a list, just as least does. Thus the following script, which redefines **minimum**, is correct:

```
import Data.List

minimum :: (Ord a) ⇒ [a] → a
minimum = head . sort
```

However, when referring to **minimum**, an ambiguity arises and it becomes necessary to use an explicit qualifier so that the name can be resolved correctly:

```
> :type minimum
<interactive>:1:0:
    Ambiguous occurrence ‘minimum’
    It could refer to either ‘Main.minimum’, defined at min.hs:4:0
                          or ‘Data.List.minimum’, imported from Data.List at min.hs:1:0-15
> :type Data.List.minimum
Data.List.minimum :: (Ord a) ⇒ [a] → a
> :type Main.minimum
Main.minimum :: (Ord a) ⇒ [a] → a
```

Sometimes it is preferrable to import the definitions from a module and to avoid that they are "merged" with all the other visible definitions in the importing module. By specifying the **qualified** keyword after **import** we instruct Haskell to import a module, but the programmer is required to explicitly qualify every name in the imported module. For example, the above redefinition of **minimum** no longer yields ambiguities if Data.List is imported qualified:

```
import qualified Data.List

minimum :: (Ord a) ⇒ [a] → a
minimum = head . Data.List.sort
```

In this way, any reference to the **minimum** function of the Data.List module must be explicitly qualified, whereas any unqualified reference is resolved to the definition in the importing module. Note that we had to explicitly qualify also the reference to **sort**:

```
> :type minimum
<interactive>:1:0:
    Ambiguous occurrence ‘minimum’
    It could refer to either ‘Main.minimum’, defined at qmin.hs:4:0
                          or ‘Data.List.minimum’, imported from Prelude
> :type Data.List.minimum
Data.List.minimum :: (Ord a) ⇒ [a] → a
```

A qualified import statement avoids potential ambiguities, but forces the programmer to qualifiy any name in the imported module. As can be seen from the Haskell library, some modules have rather long names that can become tedious and cumbersome to use in scripts. For these reasons, a qualified import statement can be followed by an alias declaration to give the imported module an alternative, shorter name. For example, the script above can also be written thus:

```
import qualified Data.List as L

minimum :: (Ord a) ⇒ [a] → a
minimum = head . L.sort
```

Now `L` is a shorter alias for the `Data.List` module, and both names can be used interchangeably:

```
> :type L.minimum
L.minimum :: (Ord a) ⇒ [a] → a
> :type Data.List.minimum
Data.List.minimum :: (Ord a) ⇒ [a] → a
```

Another possibility is that of importing only a *subset* of the names defined in a module. For instance, we could limit the import to the `sort` function in module `Data.List`:

```
import Data.List (sort)
```

The imported names are enclosed in parentheses after the import statement and separated by commas. Any name other than `sort` defined in `Data.List` is also accessible, but must be explicitly qualified:

```
> :type sort
sort :: (Ord a) ⇒ [a] → [a]
> :type Data.List.minimum
Data.List.minimum :: (Ord a) ⇒ [a] → a
```

17.3 Information hiding and abstract data types

When *defining* a module, as opposed to importing a module, it is often the case that some definitions are auxiliary and are not supposed to be used directly by the importers of the module. For example, in Section 8.4 we have defined a small library of functions for creating and manipulating fractions represented as pairs of integer numbers. Of all the defined functions, `simplFrac`, which simplifies a fraction by dividing numerator and denominator by their greatest common divisors, is used by some other functions such as `addFrac` and `mulFrac` but it may be desirable to keep it hidden within the module implementation and prevent any direct access to it from the users of the module.

The **`module`** statement can be used not only for explicitly naming the module that groups a set of related definitions, but also to limit the visibility of these definitions outside the module. In the case of fractions, we can define a proper module `Frac` like this:

```
module Frac (Fraction(..), mk, add, mul, lt) where

data Fraction = F Int Int
                deriving (Eq, Ord, Show)

mk :: Int → Int → Fraction
mk m n | n /= 0 = F m n
       | otherwise = error "Null␣denominator"

add :: Fraction → Fraction → Fraction
add (F a b) (F c d) = simpl $ F (a * d + b * c) (b * d)

mul :: Fraction → Fraction → Fraction
mul (F a b) (F c d) = simpl $ F (a * c) (b * d)

simpl :: Fraction → Fraction
simpl (F a b) = F (a ‘div‘ d) (b ‘div‘ d)
    where d = gcd a b

lt :: Fraction → Fraction → Bool
lt (F a b) (F c d) = a * d < c * b
```

The module definition at the beginning of the script simultaneously specifies the name of the module (Frac) and the list of definitions that should be *exported* from the module. In this case, all the functions except for simpl are exported. In case no list of exported definitions is given, Haskell assumes that *all* definitions are exported.

If we import the Frac module we realize that simpl is not visible:

```
> Frac.mk 1 2
F 1 2
> Frac.simpl (Frac.mk 1 2)
<interactive>:1:0: Not in scope: ‘Frac.simpl’
```

The syntax Fraction(..) means that the type Fraction along with all its constructors is also exported. This makes it possible to construct and deconstruct values of type Fraction also outside of module Frac:

```
> Frac.F 1 2 == Frac.mk 1 2
True
```

In other words, the representation of fractions is exposed. This may be undesirable in general for at least two reasons: first of all, the users of the module may depend on a particular representation of fractions, hence future changes to the Frac module may break existing code; second, users may maliciously create ill-formed data or access to information they are not supposed to see. For instance, it is perfecly possible to create a fraction with a null denominator, if one does not use the provided mk function:

```
> Frac.mk 1 0
*** Exception: Null denominator
> Frac.F 1 0
F 1 0
```

By omitting the (..) denotation next to Fraction from the module definition, one declares that the type Fraction is exported, but its representation is not. In other words, no constructor of Fraction can be directly accessed from outside the module Frac: this prevents users from creating values of type `Fraction` if not by means of the provided functions; it also prevents users from doing pattern matching of values of type `Fraction`. In this way, `Fraction` becomes an *abstract data type*. Haskell also permits a partial form of abstraction, by allowing the designer of the module to specify, in place of `..`, the list of constructors that should be exported. In general, these will be a proper subset of the constructors of the data type.

17.4 Exercises

Exercise 17.1. Define a module cobtaining all the type and value definitions of Exercise 16.4. Make sure that the internal representation of sets is hidden from the outside of the module, and that users of the module have all that is needed for manipulating sets of values.

Part VII

Advanced topics

Chapter 18

Lazy evaluation

Perhaps the most peculiar feature that distinguishes Haskell from the vast majority of programming languages is the fact that it is a *lazy* language. In this context, lazyness means that Haskell evaluates function arguments not at the time a function is applied to them, but only when the function actually uses them. This apparent twist in the evaluation strategy of Haskell has deep implications in many technical aspects of the language, but is also a valuable tool that the programmer can use for enhancing the modularity of programs.

18.1 Laziness

In Chapter 16 we have seen how to find the least element in a binary search tree:

```
least :: BTree a → a
least (Node x Empty _) = x
least (Node _ l _) = least l
```

Basically, one has to find the leftmost label in the tree, which corresponds to the least element by definition of binary search tree. The same function `least` can also be implemented thus:

```
least' :: BTree a → a
least' = head . labels
```

The idea of this alternative implementation of `least` is to collect the list of all the labels in the tree, and to return the first element of the list. If the tree is a binary search tree, the list returned by `labels` will be sorted in ascending order, thus the first element of the list effectively corresponds to the least element in the tree.

Admittedly `least'` is very elegant: it is shorter than `least`, it matches the intuition that the least element of the tree is the leftmost label, and it is solely built using existing functions (**head** and `labels`) suitably composed together. The drawback of `least'` lies, apparently, in its performance: since `labels` computes the list of *all* the labels in the tree, this means that the whole tree is visited. Conversely, `least` only needs to traverse the leftmost branch of the tree. More precisely, the complexity measured in terms of traversed nodes, is linear in the number of nodes of the tree for `least'` and linear in the depth of the tree for `least`. Thus, on a balanced tree `least` is logarithmic in the number of nodes of the tree.

Well this is true for most programming languages, but not for Haskell! Unlike other programming languages, Haskell is a *lazy language*. In practice this means that, when we apply a function to an argument, the argument is evaluated only *if* and *when* the function "uses" it. In most programming languages, the argument is always evaluated *before* the function is applied, regardless of whether the argument is actually needed by the function. For this reason such programming languages are often said to be *eager*.

The effects of lazy evaluation are usually hard to grasp, because lazyness affects the order in which expressions are turned to values and makes it hardly predictable. For this reason let us try to understand what happens within GHCi when we use `least'`. Consider the binary search tree denoted by t_1 and defined by the following equations:

```
t₁ = Node 4 t₂ t₃
t₂ = Node 2 Empty Empty
t₃ = Node 6 (Node 5 Empty Empty) (Node 7 Empty Empty)
```

and let us go through the evaluation of `least'` t_1:

```
least' t₁
  ⟹ head (labels t₁)
  ⟹ head (append (labels t₂) (append [4] (labels t₃)))
  ⟹ head (append [2] (append [4] (labels t₃)))
  ⟹ head (2 : (append [4] (labels t₃)))
  ⟹ 2
```

where for the sake of simplicity we have written **head** `(labels` t_1`)` instead of `(`**head** `. labels)` t_1 and we have used the function append instead of the concatenation operator ++. The first reduction stems from the very definition of `least'`. Now, **head** evaluates its only argument, but does so lazily: as soon as the argument has the shape of a list with head x and some arbitrary tail, it returns x (it may be useful to have a look at the definition of **head** in Section 9.3). Hence now we turn our attention to the argument of **head**, which is the expression `labels` t_1. Since t_1 is a nonempty binary tree, this unfolds to the right hand side of the equation defining `labels` (see Section 16.2), which is equivalent to:

```
append (labels t₂) (append [4] (labels t₃))
```

This expression needs further reduction, and this time we need to look at the definition of append (Example 9.4). We see that append does pattern matching over its first argument, and leaves its second argument untouched. For lazy evaluation, this means that only `labels` t_2 needs to be evaluated. After some reductions, this turns out to be the list `[2]`, since 2 is the only label within t_2. Now we can unfold the definition of append when its first argument is the nonempty list, and we obtain:

```
2 : (append [4] (labels t₃))
```

Recall that this list is the argument of **head**, and that the tail of the list is an unevaluated expression. But this is irrelevant for **head** as the head of the list is the only thing that matters. Hence the application of **head** reduces immediately to 2. In particular, the labels within t_3 are not collected, despite there was a subexpression ready to do so.

Example 18.1 (least element of a list). Suppose we need to look for the smallest element in a list of values whose type belongs to the class **Ord**. We could either write the function explicitly, as in the following script:

```
least :: (Ord a) ⇒ [a] → a
least [x] = x
least (x : xs) = min x (least xs)
```

or we could implement `least` using standard library functions:

```
least :: (Ord a) ⇒ [a] → a
least (x : xs) = foldl min x xs
```

Another possibility arises if one realizes that the least element of a list is the *first* element of the list obtained by sorting the original one. In other terms, we could also implement `least` like this:

```
least :: (Ord a) ⇒ [a] → a
least = head . sort
```

where **sort** is some arbitrary sorting function (such as `qsort` defined in Section 1.1).

Obviously the last implementation of `least`, albeit more modular, looks less efficient than the previous ones. This is because the previous implementations have a complexity that is linear in the length of the list, whereas any sorting algorithm, however efficient, is known to have a higher complexity. However, thanks to lazy evaluation, the sorting algorithm will actually run until the least element of the list has been detected. At that point **head** will immediately return the result, without further evaluating the sorted remainder of the list. ■

Example 18.2. In Example 7.2 we have seen how to write a Haskell function that mimics the behavior of a conditional expression:

```
ifFun :: Bool → a → a → a
ifFun True e _ = e
ifFun False _ e = e
```

In Section 2.4 we have also seen that, in a conditional expression, only the sub-expression in the branch corresponding to the value of the test is evaluated: if the test evaluates to **True**, only the subexpression following **then** is evaluated; if the test evaluates to **False**, only the subexpression following **else** is evaluated.

By now we can appreciate that lazy evaluation plays a fundamental role in making sure that `ifFun` is indeed equivalent to the conditional expression. In traditional programming languages, a function like `ifFun` would require *all* of its arguments to be evaluated. In this way, we would evaluate also the subexpression in the branch that does not correspond to the value of the test. In conclusion, `ifFun` works correctly thanks to Haskell's lazyness. ■

Example 18.3 (short-circuited evaluation). In Section 2.4 we have also seen that binary boolean operators (conjunction && and disjunction ||) are short-circuited, namely they evaluate their second operand only when the first one does not allow to determine the value of the whole expression. Eager programming languages that implement short-circuited evaluation of boolean operators need to treat them in a special way. In Haskell, they are no different from any other library function:

```
andFun :: Bool → Bool → Bool
andFun False _ = False
andFun True x = x

orFun :: Bool → Bool → Bool
orFun True _ = True
orFun False x = x
```

Observe that both functions require that only the first argument is evaluated, as is proved by the following expressions which do not produce a runtime error:

```
> andFun False (1 / 0 == 1)
False
```

```
> orFun True (1 / 0 == 1)
True
```

■

18.2 Infinite lists

In Example 9.1 we have defined a function `fromTo` that creates the list of integer numbers in a given range:

```
> fromTo 0 9
[0,1,2,3,4,5,6,7,8,9]
> fromTo (-5) 5
[-5,-4,-3,-2,-1,0,1,2,3,4,5]
```

Consider now the following variant of `fromTo`, which we call `from`:

```
from :: Int → [Int]
from m = m : from (m + 1)
```

This looks like an ill-founded version of `fromTo`, because `from` is a recursive function without a proper base case. This means that, when applied to an argument, it will never terminate. But if we run it in `GHCi`, somewhat surprisingly we do get something out of it:

```
> from 0
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,
47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,
69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,
91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,
109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,
125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,
^CInterrupted.
```

In fact one has to hit `^C` to stop `GHCi` printing numbers. Observe that the list of numbers we see is consistent with the evaluation of `from 0`, it appears to be the list of integer numbers starting from 0. What is most surprising at first is that it looks as if `from 0` did actually create such a list, and `GHCi` started to print every value in it. But such a list is infinite, so there is no way it may have been completely produced as the result of the evaluation of `from 0`.

The solution of this puzzle is, once again, lazyness. Haskell never tries to fully evaluate `from 0`. In fact, Haskell lazily tries not to evaluate it at all. It is `GHCi` that fires up evaluation as it tries to read and print the numbers in the list. `GHCi` asks for the first number in the list, hence `from 0` is evaluated thus:

```
  from 0 ⟹ 0 : from 1
```

The evaluation stops, `GHCi` reads the first number of the list, `0`, and prints it on the terminal. Then, it asks for the second number in the list. At this stage, Haskell realizes that `from 1` needs to be evaluated, thus:

```
  from 1 ⟹ 1 : from 2
```

The evaluation stops, `GHCi` reads the second number of the list, `1`, and prints it on the terminal. The iteration proceeds as before, *ad infinitum*. In conclusion, the creation of the list and the fact that each number in the list is printed on the terminal are *interleaved*, but this is all hidden in the evaluation strategy of Haskell.

Lazyness provides the programmer with a convenient tool for dealing with infinite structures that are created *on demand*. At first sight it might seem that this feature has little practical impact: after all, computing is all about finite structures! However, lazyness can help the programmer in writing cleaner and more modular code. The basic idea is that it becomes trivial to separate the *generation* of some data structure from the *processing* of the same structure in cases where both phases affect each other. As an example, consider the function `fromTo`:

```
fromTo :: Int → Int → [Int]
fromTo m n | m > n = []
           | otherwise = [m] ++ fromTo (m + 1) n
```

Observe that this function embeds a generation phase (the creation of a list of numbers starting from `m`) and a processing phase (the truncation of such list up to number `n`). Observe also that the two phases are mixed with each other. Consider now the function **`take`**, defined as:

```
take :: Int → [a] → [a]
take n _ | n <= 0 = []
take n (x : xs) = x : take (n - 1) xs
```

which takes the first $n \geq 0$ elements from a list. We can now express `fromTo` as a modular composition of `from` and **`take`**:

```
fromTo :: Int → Int → [Int]
fromTo m n = take (n - m) . from m
```

18.3 Case study: approximate square root

In this section we will see an application of infinite lists to the approximate computation of the square root of a number. We start by recalling Newton's method for finding the 0 of a function: if f is a function, $\hat{x}$ is a 0 of f, namely $f(\hat{x}) = 0$, and x_n is an approximation of $\hat{x}$, then

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

is a better approximation of $\hat{x}$, where f' is as usual the first derivative of f.

This gives us an iterative procedure for obtaining better and better approximations of the zero of a function. The algorithm starts with an arbitrary approximation x_0 and then continues by computing the sequence

$$x_0, x_1, x_2, \ldots$$

until two subsequent approximations x_n and x_{n+1} are found such that $|x_{n+1} - x_n| < \delta$, where δ is the tolerated approximation error.

Consider now the function

$$f(x) = x^2 - a$$

We have that $f(\hat{x}) = 0$ implies $\hat{x}^2 - a = 0$, namely $\hat{x} = \sqrt{a}$. Hence we can use Newton's method for computing the square root of a number a. In this particular case we have

$$f'(x) = 2x$$

so that the $(n+1)$-th approximation is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Now that we know how to compute arbitrarily precise approximations of the square root of a number a, let us write an Haskell function that computes *all* of them. We do so by first defining a higher-order function **`iterate`** that creates an infinite list from a starting value and a function that computes another value from the starting one:

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

En passant, observe that **`iterate`** looks very similar to `from` defined in Section ??. In fact, `from` is a specialized version of **`iterate`** where `f` is the successor function:

```
from :: Int → [Int]
from = iterate (+ 1)
```

The approximations of the square root of a number a can be obtained from `iterate (\x → (x + a /` where we use a as the first rough approximation for $\sqrt{a}$. For example, the first 5 approximations of $\sqrt{2}$ are:

```
> take 5 $ iterate (\x → (x + 2 / x) / 2) 2
[2.0,1.5,1.4166666666666665,1.4142156862745097,1.4142135623746899]
```

We have now successfully implemented the generation phase of the problem, and in doing so we have defined an auxiliary function **`iterate`** that may be useful in other contexts as well (in fact **`iterate`** is already defined in the standard library of Haskell).

We now turn the attention towards the processing phase, in which we have to find two subsequent values in the generated list that are close enough, say less than a given parameter `delta`. To this aim we can define a function `precise` that accepts `delta` and an infinite list of numbers, and returns the first number in the list that is "precise enough":

```
precise :: Float → [Float] → Float
precise delta (x : y : _) | abs (x - y) < delta = x
precise delta (_ : xs) = precise delta xs
```

For defining `asqrt` we only need to put everything together:

```
asqrt :: Float → Float → Float
asqrt delta a = precise delta $ iterate (\x → (x + a / x) / 2) a
```

A few tests confirm that `asqrt` is the function we wanted to define:

```
> asqrt 0.1 2
1.5
> asqrt 0.01 2
1.4166667
> asqrt 0.001 2
1.4142157
```

18.4 Case study: sieve of Eratosthenes

The sieve of Eratosthenes is a simple algorithm for finding all prime numbers. It works according to the following steps:

1. write down the list of all natural numbers greater than 1;
2. mark the first unmarked number in the list as prime;
3. remove any subsequent number from the list that is a multiple of the marked one;
4. repeat from step 2.

The algorithm can be described visually with the following table:

$\underline{2}$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	···
$\underline{2}$	$\underline{3}$		5		7		9		11		13		15		17	···
$\underline{2}$	$\underline{3}$		$\underline{5}$		7				11		13				17	···

where the first line contains the list of all natural numbers greater than 1. The first unmarked number in the list, 2, is marked as prime, and every subsequent even number is removed from the list. The second line shows all numbers remaining after the first iteration of the algorithm. The first unmarked number in the list, 3, is marked as prime, and every subsequent number that is multiple of 3 is removed from the list. And so on and so forth.

As a matter of facts the informal description of the algorithm can be hardly defined to be an algorithm at all, since the first step involves creating an infinite list of numbers. By now we should be confident, however, that Haskell gives us everything we need to implement the sieve of Eratosthenes exactly as has been formulated above. We already know how to create the list of all numbers starting from 2. The actual sieve is thus implemented by means of the following function:

```
sieve :: [Int] → [Int]
sieve (x : xs) = x : (sieve $ filter (not . (x ‘divides‘)) xs)
```

and the whole list of prime numbers can now be defined trivially:

```
primes :: [Int]
primes = sieve $ from 2
```

We can use **take** to extract the first n prime numbers, for an arbitrary n:

```
> take 23 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83]
```

and if we are interested in finding the smallest prime number that is greater than some given number n, this is now as simple as:

```
> head (filter (> 2048) primes)
2053
```

The problem of finding all the prime numbers that are *smaller than* some given number n is slightly more subtle. If we try to evaluate

```
> filter (< 80) primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79
^CInterrupted.
```

we see that GHCi hangs after printing a few prime numbers and we need to interrupt it with ^C. The problem lies in the fact that GHCi does not know that primes is a sequence of increasing numbers. Hence, it keeps looking for numbers that are smaller than 80 ever after it has seen 101 in the sequence. Clearly **filter** is not the right function to use. To solve this problem, we define a variant of **filter** that returns the longest prefix of a list whose elements satisfy a given predicate:

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p (x : xs) | p x = x : takeWhile p xs
takeWhile _ _ = []
```

According to the definition, **takeWhile** terminates as soon as it finds an element of the list that does not satisfy the predicate. For this reason, the following expression terminates and solves our original problem:

```
> takeWhile (< 80) primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79]
```

18.5 Exercises

Exercise 18.1. Knowing that Haskell is a lazy language, trace the whole evaluation of the following expressions and use the definitions given in this text for the library functions used below:

1. `head (from 1)`
2. `length (take 3 (from 0))`
3. `reverse [1, 2, 3]`
4. `(&&) True (1 == 0)`
5. `(&&) (1 == 0) (fibo 10 == 0)`

Exercise 18.2. Define the infinite list containing all the numbers in the Fibonacci sequence.

Exercise 18.3. Using the infinite list of all prime numbers, define a function that decomposes an integer number into a list of all its prime factors.

Exercise 18.4. Make sure to solve the previous exercise using the function `divisors`.

Chapter 19

Input and Output

Every program, to be useful, must interact with the environment in which it operates: by printing messages, by reading characters from various input devices, by reading and writing files, by sending and receiving messages over a network. We can think of all these tasks as of *actions* that, when *performed* by the program, change the environment in which the program executes. Actions are very different from the expressions we have been using so far. When an expression is evaluated, it is turned to a value. The value we obtain embeds all that we need to know about the expression, because the evaluation process has no effect on the environment. Conversely, the main consequence of performing actions resides in the effects that the actions have on the environment. In this chapter we glance at the input/output system of Haskell.

19.1 Lazyness and side effects

Until now we have been writing relatively simple Haskell scripts that could be loaded into `GHCi`. The execution of a program consisted solely in evaluating expressions. It is thus perfectly legitimate to ask whether it is possible to write a standalone Haskell program, possibly interacting with the environment in which it is executed. In most functional programming language this is accomplished by enriching the standard library of the language with functions such as **putChar** for printing a character on the terminal or **getChar** for reading a character from the input device. These functions are applied like any other function, but their execution produces so-called *side effects* on the environment.

Being Haskell a lazy language, which postpones function application until necessary, the use of functions such as **putChar** is more complicate. Consider the expression:

```
[putChar 'a', putChar 'b']
```

whose purpose would be that of printing the characters `'a'` and `'b'` and creating a list with the values returned by **putChar**, whatever they could be. In Haskell, the two applications of **putChar** are *not* evaluated before creating the list. That means, depending on how the list is used, both of `'a'` and `'b'` or only one or none of them is printed. Not even traversing the whole list is sufficient for guaranteeing that the applications are evaluated. For example

```
length [putChar 'a', putChar 'b']
```

would return 2, but the two applications would remain unevaluated since **length** does not use the elements of the list in any way.

The above discussion should make it clear that lazy evaluation and side-effects do not match very well. We can summarize this by observing that lazy evaluation makes the reduction order of an expression largely impredictable. As long as we are evaluating pure expressions, not involving side-effects, the reduction order is irrelevant as it does not affect the outcome.In expressions with side-effect, however, the order is important and we need mechanisms for specifying precisely the order in which actions should be performed.

19.2 The **IO** type constructor

The solution adopted for enriching Haskell with functions performing input/output operations consists of two features:

1. a type constructor **IO** is provided to explicitly identify actions that have input/output effects and to distinguish them from pure expressions;

2. a **do** construct to express the execution order of input/output actions and to build complex actions from simpler ones.

In Haskell, the type

```
IO a
```

denotes an input/output action that produces a value of type 'a'. When the result produced by an action is not important, the unit type `()` is used in place of 'a'. The unit type is an algebraic data type with only one constructor, which is itself denoted by `()`:

```
> :type ()
() :: ()
```

We can now have a closer look at **putChar**, starting from its type:

```
> :type putChar
putChar :: Char → IO ()
```

Namely, **putChar** is a function that accepts a character and returns an input/output action producing the unit value. Note that we have not described **putChar** as a function that *prints the character*, because that is not what **putChar** does. In fact, if we try to evaluate the expression:

```
> [putChar 'a', putChar 'b']
<interactive>:1:0:
    No instance for (Show (IO ()))
      arising from a use of ‘print’ at <interactive>:1:0-25
    Possible fix: add an instance declaration for (Show (IO ()))
    In a stmt of a 'do' expression: print it
```

we see that `GHCi` complaints about not being able to show values of type **IO** `()`, and no actual character is printed. But if **putChar** does not print the character, who does? The answer is that all the Haskell program does is to create an action of type **IO** `()`. This action is then performed by the runtime environment supporting the execution of the program. `GHCi` is aware of input/output actions, meaning that if we ask to evaluate a value of type **IO** `()`, `GHCi` will actually perform the corresponding action:

```
> putChar 'a'
aPrelude>
```

(observe the ‘a’ character that precedes the string **Prelude** in the prompt).

We are now ready to write the first Haskell script that does some real input/output, like greeting its creator:

```
greetings :: IO ()
greetings = do putChar 'H'
               putChar 'e'
               putChar 'l'
               putChar 'l'
               putChar 'o'
               putChar '\n'
```

where we have used the **do** construct to combine several input/output actions into a single compound action that, when performed, will perform all the component actions in the specified order:

```
> greetings
Hello
```

Not surprisingly, Haskell also provides a function **putStr** that generalizes **putChar** to arbitrary sequences of characters:

```
> :type putStr
putStr :: String → IO ()
> putStr "Hello\n"
Hello
```

One should realize, however, that **putStr** can be defined in terms of **putChar** and the **do** construct. This is possible because input/output actions are values:

```
putStr' :: String → IO ()
putStr' [c] = putChar c
putStr' (c : cs) = do putChar c
                      putStr' cs
```

That is, when **putStr'** is applied to a list containing a single character, it reduces to **putChar**. If the list contains two or more characters, then **putStr'** creates a compound input/output action which is made of **putChar** c where ‘c’ is the first character in the list and **putStr'** cs where cs is the tail of the list. Both **putChar** c and **putStr'** cs are values of type **IO** () and they are combined in another value of type **IO** () by the **do** construct:

```
> putStr' "Hello\n"
Hello
```

What is slightly unsatisfactory about **putStr'**, and that justifies the ' in its name, is that **putStr'** is undefined when the string to be printed is the empty one. The problem is that printing the empty string means doing nothing, so we need a way to create an action that performs no input/output. In addition, such action must be of type **IO** () because it has to match the return type of **putStr**. To create such an action we use the **return** function applied to the value ():

```
putStr :: String → IO ()
putStr [] = return ()
putStr (c : cs) = do putChar c
                     putStr cs
```

19.3 Actions that return values

The types of **putChar** and **putStr** are fairly natural if compared to the types of the corresponding functions in other programming languages. The only caveat is that in Haskell these functions do not directly write characters, but rather return an action that, when performed, writes characters. We now turn our attention to **getChar**, which is apparently more surprising:

```
> :type getChar
getChar :: IO Char
```

Observe that **getChar** is not a function. It is an input/output action that, when performed, returns a value of type **Char**, the character that has been read. While this intuition is clear, the use of **getChar** requires yet another construct. Consider the script

```
twice :: IO ()
twice = do c ← getChar
           putChar c
           putChar c
```

which defines an input/output action `twice` made of three component actions. The first action, **getChar**, reads a character from the input device (typically the keyboard) and sets ‘c’ to the character that has been read. Then, character ‘c’ is output twice with two subsequent executions of the familiar action **putChar** c. The construct

$x \leftarrow E$

can only be used withing a **do** statement. The expression E must have type **IO** t for some type t. When performed, the action generated by E produces a value of type t which is bound to the name x. This name can be used in any subsequent expression occurring in the **do** statement.

Just as we did for **putChar**, we can combine several **getChar** actions to create more complex ones, such as the following:

```
getLine :: IO String
getLine = do c ← getChar
             if c == '\n'
                 then return ""
                 else do s ← getLine
                         return (c : s)
```

Here **getLine** is a compound action that reads a sequence of characters until it finds a carriage return `'\n'`. It begins by reading a single character c. If c is the carriage return, it performs no further action and returns the empty string. Observe that we had to use **return** "" in the **then** branch and not simply "" since **getLine** has type **IO String** and not simply **String**. If c is different from `'\n'` then more characters are read by performing **getLine** recursively. These characters are bound to the name s and finally the string `c : s` is returned.

19.4 Case study: simple processing of text files

In this section we put all we know about input/output together for implementing a simple standalone Haskell program that sorts text lines. Instead of relying on our own implementation of the sorting algorithm we import the **sort** function from the Data.**List** module of the standard library:

```
import Data.List (sort)
```

The idea is to read a sequence of lines, each one terminated by the carriage return character, into a list of strings. This list can then be sorted with the **sort** function just imported. Since reading lines involves input/output operations, the value getLines that we define below must state so explicitly in its type, which is **IO** [**String**]. This indicates that getLines is an input/output action that, when performed, returns a list of strings:

```
getLines :: IO [String]
getLines = do l ← getLine
              if l == ""
                 then return []
                 else do ls ← getLines
                         return (l : ls)
```

The action getLines begins by reading a line l. If this line is empty, then the action terminates immediately by returning the empty list. If l is not the empty string, then the remaining lines are recursively collected by getLines and bound to the name ls. Then, the action returns l, the first line that has been read, composed with ls.

After the lines have been sorted, we need to print them one by one. This is accomplished by the following function putLines, which can be applied to a list of strings to generate an input/output action that prints all of them in the order in which they appear:

```
putLines :: [String] → IO ()
putLines [] = return ()
putLines (l : ls) = do putStrLn l
                       putLines ls
```

In this last code snippet we use the library function **putStrLn** which is the same as **putStr** except that it also prints a newline character after the string.

By now the program is almost finished, all we need to do is glueing everything together. We also need to instruct Haskell about the entry point of the program, namely which action should be executed first. This is accomplished by defining an input/output action called main and whose type must mandatorily be **IO** ():

```
main :: IO ()
main = do ls ← getLines
          putLines $ sort ls
```

The main action simply reads the line from the input, sorts them using **sort**, and passes them to putLines for printing.

In general every Haskell program does perform some input/output. Just like the simple program we have developed, every program is neatly separated into two domains: that of input/output actions, whose type is **IO** *t*, which interface the program with its environment, and that of pure expressions, whose type does not involve the **IO** type constructor, which is where the actual computation takes place. In the program above, getLines, putLines, and main form the interface part, whereas the actual computation is, in this case, the sole function **sort**. Of course, in most programs the computation will be far more complicated than this.

Example 19.1. The function putLines closely resembles the library function **map**, since it turns every element l in a list of strings into a corresponding action **putStrLn** l. It is indeed possible to use **map** in this context, but the result is not quite what we need:

```
> :type map putStrLn
map putStrLn :: [String] → [IO ()]
```

What we get is a list of `IO ()` actions, whereas `putLines` returns a single `IO ()` action obtained by sequencing all of these single actions. Since actions are just plain Haskell values, we can define an auxiliary sequencing function like the following:

```
sequence :: [IO ()] → IO ()
sequence [] = return ()
sequence (x : xs) = do x
                       sequence xs
```

The function `putLines` can now be defined as follows:

```
putLines :: [String] → IO ()
putLines = sequence . (map putStrLn)
```

The `Control.Monad` module of the standard library contains a few variants of the `sequence` function presented here. ■

19.5 Exercises

Exercise 19.1. Implement the Unix utility `wc`. Hint: use the library function `words`.

Bibliography

[Bac07] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. page 1977, 2007.

[BW98] Richard Bird and Philip Wadler. *Introduction to Functional Programming (2nd Edition)*. Prentice Hall PTR, April 1998.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[Hud00] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, February 2000.

[Hug90] John Hughes. Why functional programming matters. pages 17–42, 1990.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2007.

[OSG08] Bryan O'Sullivan, Donald Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, Inc., December 2008.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, March 1999.

Index