The Science of Functional Programming

A Tutorial, with Examples in Scala

by Sergei Winitzki, Ph.D.

draft version, July 3, 2019

Published by **lulu.com** 2019

Published and printed by **lulu.com**
ISBN 978-0-359-76877-6

The full source code of the book is included as an "attachment" into the PDF file `sofp.pdf`. The command `'pdftk sofp.pdf unpack_files output .'` will extract a compressed file `sofp-src.tar.bz2` to the current directory. Unpack the sources with `'tar -jxvf sofp-src.tar.bz2'`. To build the file `sofp.pdf` from the sources, run `'bash make_sofp_pdf.sh'`.

This tutorial presents the theoretical knowledge needed by practitioners of the functional programming paradigm. Detailed explanations and derivations are logically developed and accompanied by worked examples tested in the Scala interpreter, as well as exercises. Readers should have a working knowledge of basic Scala (e.g. be able to write code that reads a small text file and prints all word counts, sorted in descending order by count). Readers should also know school-level mathematics; for example, should be able to simplify the expressions $\frac{1}{x-2} - \frac{1}{x+2}$ and $\frac{d}{dx}\left((x+1)\,e^{-x}\right)$.

Contents

Preface

The goal of this book is to teach programmers how to reason mathematically about types and code, in a way that is directly relevant to software practice.

The material is presented here at medium to advanced level. It requires a certain amount of mathematical experience and is not suitable for people who are unfamiliar with school-level algebra, or unwilling to learn difficult concepts through prolonged mental concentration and effort.

The first part is introductory and may be suitable for beginners in programming. Starting from Chapter 5, the material becomes unsuitable for beginners.

The presentation in this book is self-contained. I define and explain all the required notations, concepts, and Scala language features. The emphasis is on clarity and understandability of all examples, mathematical notions, derivations, and code. To achieve a clearer presentation of the material, I use some *non-standard* notations (Appendix **??**) and terminology (Appendix **??**). The book does not assume familiarity with today's research literature in the theory of programming languages.

The main intent of this book is to explain the mathematical principles that guide the practice of functional programming – that is, help people to write code. Therefore, all mathematical developments in this book are motivated and justified by practical programming issues, and are accompanied by code examples that illustrate their usage. For example, the equational laws for standard typeclasses (functor, applicative, monad, etc.) are first motivated heuristically before deriving a set of mathematical equations and formulating the laws in more abstract terms. Each new concept or technique is explained via solved examples and exercises. Answers to exercises are not provided, but it is verified that the exercises are doable and free of errors. More difficult examples and exercises are marked by an asterisk (*).

A software engineer needs to know only a few fragments of mathematical theory; namely, the fragments that answer questions arising in the practice of functional programming. So I keep the theoretical material to the minimum; *ars longa, vita brevis*. I do not pursue mathematical generalizations beyond practical relevance or immediate pedagogical usefulness. This limits the scope of required mathematical knowledge to bare rudiments of category theory, type theory, and formal logic. For instance, I do not talk about "introduction/elimination rules", "strong normalization", "complete partial order domains", "adjoint functors", "limits", "pullbacks", or "topoi", and do not use the word "algebra", because learning these concepts will not help a functional programmer write code. Instead, I focus on practically useful material – including some rarely mentioned constructions, e.g. the "filterable functor" and "applicative contrafunctor" typeclasses.

All code examples in this book serve only for explanation and illustration, and are not optimized for performance or stack safety.

Some formatting conventions used in this book:

- Text in boldface indicates a new concept or term that is being defined. Text in italics is a logical emphasis. Example:

  An **aggregation** is a function from a list of values to a *single* value.

- Sample Scala code is written inline using a small monospaced font, such as this: `val a = "xyz"`. Longer code examples are written in separate code blocks, which may also show the output from the Scala interpreter:

```
val s = (1 to 10).toList

scala> s.product
res0: Int = 3628800
```

- Derivations of laws are written in a two-column notation where the right column contains the code and the left column indicates the property or law used to derive the expression at right. A green underline in the *previous* expression shows the part rewritten using the indicated law:

$$\begin{aligned} \text{expect to equal } \mathrm{pu}_M: \quad & \underline{\mathrm{pu}_M^{\uparrow \mathrm{Id}}} \fatsemi \mathrm{pu}_M \fatsemi \mathrm{ftn}_M \\ \text{lifting to the identity functor}: \quad & = \mathrm{pu}_M \fatsemi \underline{\mathrm{pu}_M \fatsemi \mathrm{ftn}_M} \\ \text{left identity law for } M: \quad & = \mathrm{pu}_M \quad . \end{aligned}$$

  A green underline is sometimes also used at the *last* step of the derivation, to indicate the part of the expression that resulted from the most recent rewriting.

Part I

Beginner level

1 Mathematical formulas as code. I. Nameless functions

1.1 Translating mathematics into code

1.1.1 First examples

We begin by writing Scala code for some computational tasks.

Factorial of 10 Find the product of integers from 1 to 10 (the **factorial** of 10).

First, we write a mathematical formula for the result:

$$\prod_{k=1}^{10} k \quad .$$

We can then write Scala code in a way that resembles this formula:

```
scala> (1 to 10).product
res0: Int = 3628800
```

The Scala interpreter indicates that the result is the value 3628800 of type `Int`. To define a name for this value, we use the "`val`" syntax:

```
scala> val fac10 = (1 to 10).product
fac10: Int = 3628800

scala> fac10 == 3628800
res1: Boolean = true
```

The code `(1 to 10).product` is an **expression**, which means that (1) the code can be evaluated (e.g. using the Scala interpreter) and yields a value, and (2) the code can be inserted as a part of a larger expression. For example, we could write

```
scala> 100 + (1 to 10).product + 100
res0: Int = 3629000
```

Factorial as a function Define a function that takes an integer n and computes the factorial of n.

A mathematical formula for this function can be written as

$$f(n) = \prod_{k=1}^{n} k \quad .$$

The corresponding Scala code is

```
def f(n:Int) = (1 to n).product
```

In Scala's `def` syntax, we need to specify the type of a function's argument; in this case, we write `n:Int`. In the usual mathematical notation, types of arguments are either not written at all, or written separately from the formula:

$$f(n) = \prod_{k=1}^{n} k, \quad \forall n \in \mathbb{N} \quad .$$

This indicates that n must be from the set of non-negative integers (denoted by $\mathbb{N}$ in mathematics). This is similar to specifying the type `Int` in the Scala code. So, the argument's type in the code specifies the *domain* of a function.

Having defined the function `f`, we can now apply it to an integer argument:

```
scala> f(10)
res6: Int = 3628800
```

It is an error to apply `f` to a non-integer value, e.g. to a string:

```
scala> f("abc")
<console>:13: error: type mismatch;
 found   : String("abc")
 required: Int
       f("abc")
         ^
```

1.1.2 Nameless functions

The formula and the code, as written above, both involve *naming* the function as "f". Sometimes a function does not really need a name, – for instance, if the function is used only once. I denote "nameless" mathematical functions like this:

$$x \Rightarrow (\text{some formula}) \quad .$$

Then the mathematical notation for the nameless factorial function is

$$n \Rightarrow \prod_{k=1}^{n} k \quad .$$

This reads as "a function that maps n to the product of all k where k goes from 1 to n". The Scala expression implementing this mathematical formula is

```
(n: Int) => (1 to n).product
```

This expression shows Scala's syntax for a **nameless** function. Here,

```
n: Int
```

is the function's **argument**, while

```
(1 to n).product
```

is the function's **body**. The arrow symbol `=>` separates the argument from the body.[1]

Functions in Scala (whether named or nameless) are treated as values, which means that we can also define a Scala value as

```
scala> val fac = (n: Int) => (1 to n).product
fac: Int => Int = <function1>
```

We see that the value `fac` has the type `Int => Int`, which means that the function takes an integer (`Int`) argument and returns an integer result value. What is the value of the function `fac` *itself*? As

[1] In Scala, the two ASCII characters `=>` and the single Unicode character ⇒ have the same meaning. I use the symbol ⇒ (pronounced "maps to") in this book. However, when doing calculations *by hand*, I tend to write → instead of ⇒ since it is faster. Several programming languages, such as OCaml and Haskell, use the symbols -> or the Unicode equivalent, →, for the function arrow.

we have just seen, the Scala interpreter prints `<function1>` as the "value" of `fac`. An alternative Scala interpreter[2] called `ammonite` prints something like this,

```
scala@ val fac = (n: Int) => (1 to n).product
fac: Int => Int = ammonite.$sess.cmd0$$$Lambda$1675/2107543287@1e44b638
```

This seems to indicate some identifying number, or perhaps a memory location.

I usually imagine that a "function value" represents a block of compiled machine code, – code that will actually run and evaluate the function's body when the function is applied to its argument.

Once defined, a function can be applied to an argument like this:

```
scala> fac(10)
res1: Int = 3628800
```

However, functions can be used without naming them. We can directly apply a nameless factorial function to an integer argument `10` instead of writing `fac(10)`:

```
scala> ((n: Int) => (1 to n).product)(10)
res2: Int = 3628800
```

One would not often write code like this because there is no advantage in creating a nameless function and then applying it right away to an argument. This is so because we can evaluate the expression

```
((n: Int) => (1 to n).product)(10)
```

by substituting `10` instead of `n` in the function body, which gives us

```
(1 to 10).product
```

If a nameless function uses the argument several times, for example

```
((n: Int) => n*n*n + n*n)(12345)
```

it is still better to substitute the argument and to eliminate the nameless function. We could have written

```
12345*12345*12345 + 12345*12345
```

but, of course, we want to avoid repeating the value `12345`. To achieve that, we may define `n` as a value in an **expression block** like this:

```
scala> { val n = 12345; n*n*n + n*n }
res3: Int = 322687002
```

Defined in this way, the value `n` is visible only within the expression block. Outside the block, another value named `n` could be defined independently of this `n`. For this reason, the definition of `n` is called a **locally scoped** definition.

Nameless functions are most useful when they are themselves arguments of other functions, as we will see next.

Example: prime numbers Let us define a function that takes an integer argument n and determines whether n is a prime number.

A simple mathematical formula for this function can be written as

$$\text{is\_prime}\,(n) = \forall k \in [2, n-1] \ : \ n \neq 0 \bmod k \quad . \tag{1.1}$$

[2] https://ammonite.io/

This formula has two clearly separated parts: first, a range of integers from 2 to $n-1$, and second, a requirement that all these integers should satisfy a given condition, $n \neq 0 \bmod k$. Formula (1.1) is translated into Scala code as

```
def is_prime(n: Int) = (2 to n-1).forall(k => n % k != 0)
```

In this code, the two parts of the mathematical formula are implemented in a way that is closely similar to the mathematical notation, except for the arrow after k.

We can now apply the function `is_prime` to some integer values:

```
scala> is_prime(12)
res3: Boolean = false

scala> is_prime(13)
res4: Boolean = true
```

As we can see from the output above, the function `is_prime` returns a value of type `Boolean`. Therefore, the function `is_prime` has type `Int => Boolean`.

A function that returns a `Boolean` value is called a **predicate**.

In Scala, it is optional – but strongly recommended – to specify the return type of named functions. The required syntax looks like this,

```
def is_prime(n: Int): Boolean =
  (2 to n-1).forall(k => n % k != 0)
```

However, we do not need to specify the type `Int` for the argument `k` of the nameless function `k => n % k != 0`. This is because the Scala compiler knows that `k` is going to iterate over the *integer* elements of the range `(2 to n-1)`, which effectively forces `k` to be of type `Int`.

1.1.3 Nameless functions and bound variables

The code for `is_prime` differs from the mathematical formula (1.1) in two ways.

One difference is that the interval $[2, n-1]$ is in front of `forall`. To understand this, look at the ways Scala allows programmers to define syntax.

The Scala syntax such as `(2 to n-1).forall(k => ...)` means to apply a function called `forall` to *two* arguments: the first argument is the range `(2 to n-1)`, and the second argument is the nameless function `(k => ...)`. In Scala, the **infix** syntax `x.f(z)`, or equivalently `x f z`, means that a function `f` is applied to its *two* arguments, `x` and `z`. In the ordinary mathematical notation, this would be $f(x, z)$. Infix notation is often easier to read and is also widely used in mathematics, for instance when we write $x + y$ rather than something like $plus(x, y)$.

A single-argument function could be also defined with infix notation, and then the syntax is `x.f`, as in the expression `(1 to n).product` we have seen before.

The infix methods `.product` and `.forall` are already provided in the Scala standard library, so it is natural to use them. If we want to avoid the infix syntax, we could define a function `for_all` with two arguments and write code like this,

```
for_all(2 to n-1, k => n % k != 0)
```

This would have brought the syntax somewhat closer to the formula (1.1).

However, there still remains the second difference: The symbol k is used as an *argument* of a nameless function `(k => n % k != 0)` in the Scala code, – while the mathematical notation, such as

$$\forall k \in [2, n-1] \ : \ n \neq 0 \bmod k \quad ,$$

does not seem to involve any nameless functions. Instead, the mathematical formula defines the symbol k that "goes over the range $[2, n-1]$," as one might say. The symbol k is then used for

writing the predicate $n \neq 0 \mod k$.

However, let us investigate the role of the symbol k more closely.

The symbol k is a mathematical variable that is actually defined *only inside* the expression "$\forall k : ...$" and makes no sense outside that expression. This becomes clear by looking at Eq. (1.1): The variable k is not present in the left-hand side and could not possibly be used there. The name "k" is defined only in the right-hand side, where it is first mentioned as the arbitrary element $k \in [2, n-1]$ and then used in the sub-expression "... mod k".

So, the mathematical notation

$$\forall k \in [2, n-1] \ : \ n \neq 0 \bmod k$$

gives two pieces of information: first, we are examining all values from the given range; second, we chose the name k for the values from the given range, and for each of those k we need to evaluate the expression $n \neq 0 \bmod k$, which is a certain given *function of* k that returns a `Boolean` value. Translating the mathematical notation into code, it is therefore natural to use a nameless function of k,

$$k \Rightarrow n \neq 0 \bmod k \quad ,$$

and to write Scala code that applies this nameless function to each element of the range $[2, n-1]$ and then requires that all result values be `true`:

```
(2 to n-1).forall(k => n % k != 0)
```

Just as the mathematical notation defines the variable k only in the right-hand side of Eq. (1.1), the argument `k` of the nameless Scala function `k => n % k != 0` is defined only within that function's body and cannot be used in any code outside the expression `n % k != 0`.

Variables that are defined only inside an expression and are invisible outside are called **bound variables**, or "variables bound in an expression". Variables that are used in an expression but are defined outside it are called **free variables**, or "variables occurring free in an expression". These concepts apply equally well to mathematical formulas and to Scala code. For example, in the mathematical expression $k \Rightarrow n \neq 0 \bmod k$ (which is a nameless function), the variable k is bound (it is defined only within that expression) but the variable n is free (it is defined outside that expression).

The main difference between free and bound variables is that bound variables can be *locally renamed* at will, unlike free variables. To see this, consider that we could rename k to z and write instead of Eq. (1.1) an equivalent definition

$$\text{is\_prime}\,(n) = \forall z \in [2, n-1] \ : \ n \neq 0 \bmod z \quad ,$$

or in Scala code,

```
(2 to n-1).forall(z => n % z != 0)
```

In the nameless function `k => n % k != 0`, the argument `k` may be renamed to `z` or to anything else, without changing the value of the entire program. No code outside this expression needs to be changed after renaming `k` to `z`. But the value `n` is defined outside and thus cannot be renamed locally (i.e. only within the sub-expression). If, for any reason, we wanted to rename `n` in the sub-expression `k => n % k != 0`, we would also need to change every place in the code that defines and uses `n` *outside* that expression, or else the program would become incorrect.

Mathematical formulas use bound variables in various constructions such as $\forall k : p(k)$, $\exists k : p(k)$, $\sum_{k=a}^{b} f(k)$, $\int_0^1 k^2 dk$, $\lim_{n\to\infty} f(n)$, and $\text{argmax}_k f(k)$. When translating mathematical expressions into code, we need to recognize the presence of bound variables, which the mathematical notation does not make quite so explicit. For each bound variable, we need to create a nameless function whose argument is that variable, e.g. $k \Rightarrow p(k)$ or $k \Rightarrow f(k)$ for the examples just shown. Only then will our code correctly reproduce the behavior of bound variables in mathematical expressions.

As an example, the mathematical formula

$$\forall k \in [1, n] : p(k) \quad ,$$

has a bound variable k and is translated into Scala code as

```
(1 to n).forall(k => p(k))
```

At this point we can apply the following simplification trick to this code. The nameless function $k \Rightarrow p(k)$ does exactly the same thing as the (named) function p: It takes an argument, which we may call k, and returns $p(k)$. So, we can simplify the Scala code above to

```
(1 to n).forall(p)
```

The simplification of $x \Rightarrow f(x)$ to just f is always possible for functions f of a single argument.[3]

1.2 Aggregating data from sequences

Consider the task of finding how many even numbers there are in a given list L of integers. For example, the list $[5, 6, 7, 8, 9]$ contains *two* even numbers: 6 and 8.

A mathematical formula for this task can be written like this,

$$\text{count\_even}(L) = \sum_{k \in L} \text{is\_even}(k) \quad ,$$

$$\text{is\_even}(k) = \begin{cases} 1 & \text{if } k = 0 \bmod 2 \\ 0 & \text{otherwise} \end{cases} \quad .$$

Here we defined a helper function `is_even` in order to write more easily a formula for `count_even`. In mathematics, complicated formulas are often split into simpler parts by defining helper expressions.

We can write the Scala code similarly. We first define the helper function `is_even`; the Scala code can be written in the style quite similar to the mathematical formula:

```
def is_even(k: Int): Int = (k % 2) match {
  case 0 => 1 // First, check if it is zero.
  case _ => 0 // The underscore matches everything else.
}
```

For such a simple computation, we could also write shorter code using a nameless function,

```
val is_even = (k: Int) => if (k % 2 == 0) 1 else 0
```

Given this function, we now need to translate into Scala code the expression $\sum_{k \in L} \text{is\_even}(k)$. We can represent the list L using the data type `List[Int]` from the Scala standard library.

To compute $\sum_{k \in L} \text{is\_even}(k)$, we must apply the function `is_even` to each element of the list L, which will produce a list of some (integer) results, and then we will need to add all those results together. It is convenient to perform these two steps separately. This can be done with the functions `.map` and `.sum`, defined in the Scala standard library as infix methods for the data type `List`.

The method `.sum` is similar to `.product` and is defined for any `List` of numerical types (`Int`, `Float`, `Double`, etc.). It computes the sum of all numbers in the list:

```
scala> List(1, 2, 3).sum
res0: Int = 6
```

[3] Certain features of Scala allow programmers to write code that looks like `f(x)` but actually involves additional implicit or default arguments of the function `f`, or an implicit conversion for its argument `x`. In those cases, replacing the code `x => f(x)` by just `f` may fail to compile in Scala. But these complications do not arise when working with simple functions.

The method `.map` needs more explanation. This method takes a *function* as its second argument, applies that function to each element of the list, and puts all the results into a *new* list, which is then returned as the result value:

```
scala> List(1, 2, 3).map(x => x*x + 100*x)
res1: List[Int] = List(101, 204, 309)
```

In this example, the argument of `.map` is the nameless function $x \Rightarrow x^2 + 100x$. This function is repeatedly applied by `.map` to transform each of the values from a given list, creating a new list as a result.

It is equally possible to define the transforming function separately, give it a name, and then pass it as the argument to `.map`:

```
scala> def func1(x: Int): Int = x*x + 100*x
func1: (x: Int)Int

scala> List(1, 2, 3).map(func1)
res2: List[Int] = List(101, 204, 309)
```

Usually, short and simple functions are defined inline, while longer functions are given a name and defined separately.

An infix method, such as `.map`, can be also used with a "dotless" syntax:

```
scala> List(1, 2, 3) map func1
res3: List[Int] = List(101, 204, 309)
```

If the transforming function `func1` is used only once, and especially for a simple operation such as $x \Rightarrow x^2 + 100x$, it is easier to work with a nameless function.

We can now combine the methods `.map` and `.sum` to define `count_even`:

```
def count_even(s: List[Int]) = s.map(is_even).sum
```

This code can be also written using a nameless function instead of `is_even`:

```
def count_even(s: List[Int]): Int =
   s
    .map { k => if (k % 2 == 0) 1 else 0 }
    .sum
```

It is customary in Scala to use infix methods when chaining several operations. For instance `s.map(...).sum` means first apply `s.map(...)`, which returns a *new* list, and then apply `.sum` to that list. To make the code more readable, I put each of the chained methods on a new line.

To test this code, let us run it in the Scala interpreter. In order to let the interpreter work correctly with code entered line by line, the dot character needs to be at the end of the line. The interpreter will automatically insert the visual continuation characters. (In a compiled code, the dots can be at the beginning of the lines since the compiler reads the entire code at once.)

```
scala> def count_even(s: List[Int]): Int =
     |   s .
     |    map { k => if (k % 2 == 0) 1 else 0 } .
     |    sum
count_even: (s: List[Int])Int

scala> count_even(List(1,2,3,4,5))
res0: Int = 2

scala> count_even( List(1,2,3,4,5).map(x => x * 2) )
res1: Int = 5
```

Note that the Scala interpreter prints the types differently for functions defined using `def`. It prints `(s: List[Int])Int` for the function type that one would normally write as `List[Int] => Int`.

1.3 Filtering and truncating sequences

In addition to the methods `.sum`, `.product`, `.map`, `.forall` that we have already seen, the Scala standard library defines many other useful methods. We will now take a look at using the methods `.max`, `.min`, `.exists`, `.size`, `.filter`, and `.takeWhile`.

The methods `.max`, `.min`, and `.size` are self-explanatory:

```
scala> List(10, 20, 30).max
res2: Int = 30

scala> List(10, 20, 30).min
res3: Int = 10

scala> List(10, 20, 30).size
res4: Int = 3
```

The methods `.forall`, `.exists`, `.filter`, and `.takeWhile` require a predicate as an argument. The `.forall` method returns `true` iff the predicate is true on all values in the list; the `.exists` method returns `true` iff the predicate holds (returns `true`) for at least one value in the list. These methods can be written as mathematical formulas like this:

$$\text{forall}\,(S, p) = \forall k \in S : p(k) = \text{true}$$
$$\text{exists}\,(S, p) = \exists k \in S : p(k) = \text{true}$$

However, there is no mathematical notation for operations such as "removing elements from a list", so we will focus on the Scala syntax for these functions.

The `.filter` method returns a *new list* that contains only the values for which the predicate returns `true`:

```
scala> List(1, 2, 3, 4, 5).filter(k => k % 3 != 0)
res5: List[Int] = List(1, 2, 4, 5)
```

The `.takeWhile` method truncates a given list, returning a *new list* with the initial portion of values from the original list for which predicate keeps being `true`:

```
scala> List(1, 2, 3, 4, 5).takeWhile(k => k % 3 != 0)
res6: List[Int] = List(1, 2)
```

In all these cases, the predicate's argument `k` must be of the same type as the elements in the list. In the examples shown above, the elements are integers (i.e. the lists have type `List[Int]`), therefore `k` must be of type `Int`.

The methods `.max`, `.min`, `.sum`, and `.product` are defined on lists of *numeric types*, such as `Int`, `Double`, and `Long`. The other methods are defined on lists of all types.

Using these methods, we can solve many problems that involve transforming and aggregating data stored in lists (as well as in arrays, sets, or other similar data structures). A **transformation** is a function from a list of values to another list of values; examples of transformation functions are `.filter` and `.map`. An **aggregation** is a function from a list of values to a *single* value; examples of aggregation functions are `.max` and `.sum`.

Writing programs by chaining together various functions of transformation and aggregation is known as programming in the **map/reduce style**.

1.4 Solved examples

1.4.1 Aggregation

Example 1.4.1.1 Improve the code for `is_prime` by limiting the search to $k^2 \leq n$:

$$\text{is\_prime}\,(n) = \forall k \in [2, n-1] \text{ such that } k^2 \leq n \ : \ n \neq 0 \bmod k \quad .$$

Solution: Use `.takeWhile` to truncate the initial list when $k^2 \leq n$ becomes false:

```
def is_prime(n: Int): Boolean =
  (2 to n-1)
    .takeWhile(k => k*k + 1 < n)
    .forall(k => n % k != 0)
```

Example 1.4.1.2 Compute $\prod_{k\in[1,10]} |\sin(k+2)|$.
Solution:

```
(1 to 10)
  .map(k => math.abs(math.sin(k + 2)))
  .product
```

Example 1.4.1.3 Compute $\sum_{k\in[1,10];\ \cos k>0} \sqrt{\cos k}$.
Solution:

```
(1 to 10)
  .filter(k => math.cos(k) > 0)
  .map(k => math.sqrt(math.cos(k)))
  .sum
```

It is safe to compute $\sqrt{\cos k}$, because we have first filtered the list by keeping only values k for which $\cos k > 0$:

```
scala> (1 to 10).toList.
  filter(k => math.cos(k) > 0).map(x => math.cos(x))
res0: List[Double] = List(0.5403023058681398, 0.28366218546322625, 0.9601702866503661,
    0.7539022543433046)
```

Example 1.4.1.4 Compute the average of a non-empty list of type `List[Double]`,

$$\text{average}\,(s) = \frac{1}{n}\sum_{i=0}^{n-1} s_i \quad .$$

Solution: We need to divide the sum by the length of the list:

```
scala> def average(s: List[Double]): Double = s.sum / s.size
average: (s: List[Double])Double

scala> average(List(1.0, 2.0, 3.0))
res0: Double = 2.0
```

Example 1.4.1.5 Given n, compute the Wallis product truncated up to $\frac{2n}{2n+1}$:

$$\text{wallis}\,(n) = \frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\cdots\frac{2n}{2n+1} \quad .$$

Solution: We will define the helper function `wallis_frac(i)` that computes the i^{th} fraction. The method `.toDouble` converts integers to `Double` numbers.

```
def wallis_frac(i: Int): Double =
   (2*i).toDouble / (2*i - 1)*(2*i)/(2*i + 1)

def wallis(n: Int) = (1 to n).map(wallis_frac).product

scala> math.cos(wallis(10000)) // Should be close to 0.
res0: Double = 3.9267453954401036E-5

scala> math.cos(wallis(100000)) // Should be even closer to 0.
res1: Double = 3.926966362362075E-6
```

The limit of the Wallis product is $\frac{\pi}{2}$, so the cosine of `wallis(n)` tends to zero in the limit of large n.

Example 1.4.1.6 Another known series related to π is

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} \quad .$$

Define a function of n that computes a partial sum of this series until $k = n$. Compute the result for a large value of n and compare with the limit value.

Solution:

```
def euler_series(n: Int): Double =
  (1 to n).map(k => 1.0/k/k).sum

scala> euler_series(100000)
res0: Double = 1.6449240668982423

scala> val pi = 4*math.atan(1)
pi: Double = 3.141592653589793

scala> pi*pi/6
res1: Double = 1.6449340668482264
```

Example 1.4.1.7 Check numerically the infinite product formula

$$\prod_{k=1}^{\infty} \left(1 - \frac{x^2}{k^2}\right) = \frac{\sin \pi x}{\pi x} \quad .$$

Solution: We will compute this product up to $k = n$ for $x = 0.1$ and a large value of n, say $n = 10^5$, and compare with the right-hand side:

```
def sine_product(n: Int, x: Double): Double =
  (1 to n).map(k => 1.0 - x*x/k/k).product

scala> sine_product(n = 100000, x = 0.1) // Arguments may be named, for clarity.
res0: Double = 0.9836317414461351

scala> math.sin(pi*0.1)/pi/0.1
res1: Double = 0.9836316430834658
```

Example 1.4.1.8 Define a function p that takes a list of integers and a function `f: Int => Int`, and returns the largest value of $f(x)$ among all x in the list.

Solution:

```
def p(s: List[Int], f: Int => Int): Int = s.map(f).max
```

Here is an example test for this function:

```
scala> p(List(2, 3, 4, 5), x => 60 / x)
res0: Int = 30
```

1.4.2 Transformation

Example 1.4.2.1 Given a list of lists, `s: List[List[Int]]`, select the inner lists of size at least 3. The result must be again of type `List[List[Int]]`.

Solution: To "select the inner lists" means to compute a *new* list containing only the desired inner lists. We use `.filter` on the outer list `s`. The predicate for the filter is a function that takes an inner list and returns `true` if the size of that list is at least 3. Write the predicate as a nameless function, `t => t.size >= 3`:

```
def f(s: List[List[Int]]): List[List[Int]] =
  s.filter(t => t.size >= 3)

scala> f(List( List(1,2), List(1,2,3), List(1,2,3,4) ))
res0: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3, 4))
```

The predicate in the argument of `.filter` is a nameless function `t => t.size >= 3` whose argument `t` is of type `List[Int]`. The Scala compiler deduces the type of `t` from the code; no other type would work with the way we use `.filter` on a *list of lists* of integers.

Example 1.4.2.2 Find all integers $k \in [1, 10]$ such that there are at least three different integers j, where $1 \leq j \leq k$, each j satisfying the condition $j^2 > 2k$.

Solution:

```
scala> (1 to 10).toList.filter(k => (1 to k).
     |  filter(j => j*j > 2*k).size >= 3)
res0: List[Int] = List(6, 7, 8, 9, 10)
```

The argument of the outer `.filter` is a nameless function that itself uses another `.filter`. The inner expression

```
(1 to k).filter(j => j*j > 2*k).size >= 3
```

computes the list of j's that satisfy the condition $j^2 > 2k$, and then compares the size of that list with 3 and so imposes the requirement that there should be at least 3 values of j. We can see how the Scala code closely follows the mathematical formulation of the problem.

1.5 Summary

Table 1.1 shows how mathematical formulas are translated into code.

What problems can one solve with this knowledge?

- Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges, such as $\sum_{k=1}^{n} f(k)$ etc.
- Transform and aggregate data from lists using `.map`, `.filter`, `.sum`, and other methods from the Scala standard library.

What are examples of problems that are not solvable with these tools?

- Example 1: Compute the smallest $n \geq 1$ such that

$$f(f(f(...f(0)...) > 1000 \quad ,$$

Mathematical notation	Scala code
$x \Rightarrow \sqrt{x^2+1}$	`x ⇒ math.sqrt(x*x + 1)`
list $[1, 2, ..., n]$	`(1 to n)`
list $[f(1), ..., f(n)]$	`(1 to n).map(k ⇒ f(k))`
$\sum_{k=1}^{n} k^2$	`(1 to n).map(k ⇒ k*k).sum`
$\prod_{k=1}^{n} f(k)$	`(1 to n).map(f).product`
$\forall k$ such that $1 \leq k \leq n : p(k)$ holds	`(1 to n).forall(k ⇒ p(k))`
$\exists k, 1 \leq k \leq n$ such that $p(k)$ holds	`(1 to n).exists(k ⇒ p(k))`
$\sum_{k \in S \text{ such that } p(k) \text{ holds}} f(k)$	`s.filter(p).map(f).sum`

Table 1.1: Translating mathematics into code.

where the given function f is applied n times.

- Example 2: Given a list s of numbers, compute the list r of running averages:

$$r_n = \frac{1}{n} \sum_{k=0}^{n-1} s_k \quad .$$

- Example 3: Perform binary search over a sorted list of integers.

These computations involve *mathematical induction*, which we have not yet learned to translate into code in the general case.

Library functions we have seen so far, such as `.map` and `.filter`, implement a restricted class of iterative operations on lists: namely, operations that process each element of a given list independently and accumulate results. For instance, when computing `s.map(f)`, the number of function applications is given by the size of the initial list. However, Example 1 requires applying a function f repeatedly until a given condition holds – that is, repeating for an *initially unknown* number of times. So it is impossible to write an expression containing `.map`, `.filter`, `.takeWhile`, etc., that solves Example 1. We could write the solution of Example 1 as a formula by using mathematical induction, but we have not yet seen how to implement that in Scala code.

Similarly, Example 2 defines a new list r from s by induction,

$$r_0 = s_0 \quad ; \qquad r_i = s_i + r_{i-1}, \forall i > 0 \quad .$$

However, operations such as `.map` and `.filter` cannot compute r_i depending on the value of r_{i-1}.

Example 3 defines the search result by induction: the list is split in half, and search is performed by inductive hypothesis in the half that contains the required value. This computation requires an initially unknown number of steps.

Chapter 2 explains how to solve these problems by translating mathematical induction into code using recursion.

1.6 Exercises

1.6.1 Aggregation

Exercise 1.6.1.1 Machin's formula converges to π faster than Example 1.4.1.5:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad ,$$
$$\arctan \frac{1}{n} = \frac{1}{n} - \frac{1}{3}\frac{1}{n^3} + \frac{1}{5}\frac{1}{n^5} - ... = \sum_{k=1}^{\infty} \frac{(-1)^k}{2k+1} n^{-2k-1} \quad .$$

Implement a function that computes the series for $\arctan \frac{1}{n}$ up to a given number of terms, and compute an approximation of π using this formula. Show that about 12 terms of the series are already sufficient for a full-precision `Double` approximation of π.

Exercise 1.6.1.2 Using the function `is_prime`, check numerically the Euler product formula for the Riemann zeta function $\zeta(4)$; it is known that $\zeta(4) = \frac{\pi^4}{90}$:

$$\prod_{k \geq 2;\ k \text{ is prime}} \frac{1}{1 - p^{-4}} = \frac{\pi^4}{90} \quad .$$

1.6.2 Transformation

Exercise 1.6.2.1 Define a function `add_20` of type `List[List[Int]] => List[List[Int]]` that adds 20 to every element of every inner list. A sample test:

```
scala> add_20( List( List(1), List(2, 3) ) )
res0: List[List[Int]] = List(List(21), List(22, 23))
```

Exercise 1.6.2.2 An integer n is called a "3-factor" if it is divisible by only three different integers j such that $2 \leq j < n$. Compute the set of all "3-factor" integers n among $n \in [1, ..., 1000]$.

Exercise 1.6.2.3 Given a function `f: Int => Boolean`, an integer n is called a "3-f" if there are only three different integers $j \in [1, ..., n]$ such that $f(j)$ returns `true`. Define a function that takes f as an argument and returns a sequence of all "3-f" integers among $n \in [1, ..., 1000]$. What is the type of that function? Implement Exercise 1.6.2.2 using that function.

Exercise 1.6.2.4 Define a function `see100` of type `List[List[Int]] => List[List[Int]]` that selects only those inner lists whose largest value is at least 100. Test with:

```
scala> see100( List( List(0, 1, 100), List(60, 80), List(1000) ) )
res0: List[List[Int]] = List(List(0, 1, 100), List(1000))
```

Exercise 1.6.2.5 Define a function of type `List[Double] => List[Double]` that "normalizes" the list: finds the element having the largest absolute value and, if that value is nonzero, divides all elements by that factor and returns a new list; otherwise returns the original list.

1.7 Discussion

1.7.1 Functional programming as a paradigm

Functional programming (FP) is a **paradigm** of programming, – that is, an approach that guides programmers to write code in specific ways, for a wide range of programming tasks.

The main principle of FP is to write code *as a mathematical expression or formula*. This approach allows programmers to derive code through logical reasoning rather than through guessing, – similarly to how books on mathematics reason about mathematical formulas and derive results systematically, without guessing or "debugging." Similarly to mathematicians and scientists who reason about formulas, functional programmers can *reason about code* systematically and logically, based on rigorous principles. This is possible only because code is written as a mathematical formula.

Mathematical intuition is backed by the vast experience accumulated while working with data over thousands of years of human history. It took centuries to invent flexible and powerful notation such as $\forall k \in S : p(k)$ and to develop the corresponding rules of reasoning. Functional programmers are fortunate to have at their disposal such a superior reasoning tool.

As we have seen, the Scala code for certain computational tasks corresponds quite closely to mathematical formulas. (Scala conventions and syntax, of course, require programmers to spell out certain things that the mathematical notation leaves out.) Just as in mathematics, large code expressions may be split into parts in a suitable way, so that the parts can be easily reused, flexibly composed together, and written independently from each other. The FP community has developed a toolkit of functions (such as `.map`, `.filter`, etc.) that proved especially useful in real-life programming, although many of them are not standard in mathematical literature.

Mastering FP involves practicing to reason about programs as formulas, building up the specific kind of applied mathematical intuition, familiarizing oneself with concepts adapted to programming needs, and learning how to translate the mathematics into code in various cases. The FP community has discovered a number of specific design patterns, founded on mathematical principles but driven by practical necessities of programming rather than by the needs of academic mathematics. This book explains the required mathematical principles in detail, developing them through intuition and practical coding tasks.

1.7.2 Functional programming languages

It is possible to apply the FP paradigm while writing code in any programming language. However, some languages lack certain features that make FP techniques much easier to use in practice. For example, in a language such as Python or Ruby, one can productively use only a limited number of FP idioms, such as the map/reduce operations. More advanced FP constructions are impractical in these languages because the required code becomes too hard to read, and mistakes are too easy to make, which negates the advantage of easier reasoning about functional programs.

Some programming languages, such as Haskell and OCaml, were designed specifically for advanced use in the FP paradigm. Other languages, such as ML, F#, Scala, Swift, Elm, and PureScript, have different design goals but still support enough FP features to be considered FP languages. I will be using Scala in this book, but exactly the same constructions could be implemented in other FP languages in a similar way. At the level of detail needed in this book, the differences between languages such as ML, OCaml, Haskell, F#, Scala, Swift, Elm, or PureScript will not play a significant role.

1.7.3 The mathematical meaning of variables

The usage of variables in functional programming closely corresponds to how mathematical literature uses variables. In mathematics, **variables** are used first of all as *arguments* of functions; e.g. the formula

$$f(x) = x^2 + x$$

contains the variable x and defines a function f that takes x as its argument (to be definite, let us assume that x is an integer) and computes the value $x^2 + x$. The body of the function is the expression $x^2 + x$.

Mathematics has the convention that a variable, such as x, does not change its value within a formula. Indeed, there is no mathematical notation even to talk about "changing" the value of x *inside* the formula $x^2 + x$. It would be quite confusing if a mathematics textbook said "before adding the last x in the formula $x^2 + x$, we change that x by adding 4 to it". If the "last x" in $x^2 + x$ needs to have a 4 added to it, a mathematics textbook will just write the formula $x^2 + x + 4$.

Arguments of nameless functions are also immutable. Consider, for example,

$$f(n) = \sum_{k=0}^{n} k^2 + k \quad .$$

Here, n is the argument of the function f, while k is the argument of the nameless function $k \Rightarrow k^2 + k$. Neither n nor k can be "modified" in any sense within the expressions where they are used. The symbols k and n stand for some integer values, and these values are immutable. Indeed, it is meaningless to say that we "modified the integer 4". In the same way, we cannot modify k.

So, a variable in mathematics remains constant *within the expression* where it is defined; in that expression, a variable is essentially a *named constant* value. Of course, a function f can be applied to different values x, to compute a different result $f(x)$ each time. However, a given value of x will remain unmodified within the body of the function f while $f(x)$ is being computed.

Functional programming adopts this convention from mathematics: variables are immutable named constants. (Scala also has *mutable* variables, but we will not need to consider them in this book.)

In Scala, function arguments are immutable within the function body:

```
def f(x: Int) = x * x + x // Can't modify x here.
```

The *type* of each mathematical variable (such as integer, string, etc.) is also fixed in advance. In mathematics, each variable is a value from a specific set, known in advance (the set of all integers, the set of all strings, etc.). Mathematical formulas such as $x^2 + x$ do not express any "checking" that x is indeed an integer and not, say, a string, before starting to evaluate $x^2 + x$.

Functional programming adopts the same view: Each argument of each function must have a **type**, which represents *the set of possible allowed values* for that function argument. The programming language's compiler will automatically check the types of all arguments *before* the program runs. A program that calls functions on arguments of incorrect types will not compile.

The second usage of **variables** in mathematics is to denote expressions that will be reused. For example, one writes: let $z = \frac{x-y}{x+y}$ and now compute $\cos z + \cos 2z + \cos 3z$. Again, the variable z remains immutable, and its type remains fixed.

In Scala, this construction (defining an expression to be reused later) is written with the "`val`" syntax. Each variable defined using "`val`" is a named constant, and its type and value are fixed at the time of definition. Type annotations for "`val`"s are optional in Scala: for instance we could write

```
val x: Int = 123
```

or we could omit the type annotation `:Int` and write more concisely

```
val x = 123
```

because it is clear that this `x` is an integer. However, when types are complicated, it helps to write them out. If we do so, the compiler will check that the types match correctly and give an error message whenever wrong types are used:

```
scala> val x: Int = "123" // A String instead of an Int.
<console>:11: error: type mismatch;
 found   : String("123")
 required: Int
       val x: Int = "123" // A String instead of an Int.
```

1.7.4 Iteration without loops

Another distinctive feature of the FP paradigm is the absence of explicit loops.

Iterative computations are ubiquitous in mathematics; as an example, consider the formula for the standard deviation estimated from a sample,

$$\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\sum_{j=1}^{n}s_i s_j - \frac{1}{n\,(n-1)}\left(\sum_{i=1}^{n}s_i\right)^2} \quad .$$

To compute these expressions, we need to iterate over values of i and j. And yet, no mathematics textbook uses "loops" or says "now repeat this formula ten times". Indeed, it would be pointless to evaluate a formula such as $x^2 + x$ ten times, or to "repeat" an equation such as

$$(x-1)\,(x^2+x+1) = x^3 - 1 \quad .$$

Instead of loops, mathematicians write *expressions* such as $\sum_{i=1}^{n} s_i$, where symbols such as $\sum_{i=1}^{n}$ or $\prod_{i=1}^{n}$ denote iterative computations. Such computations are defined using mathematical induction. The functional programming paradigm has developed rich tools for translating mathematical induction into code. In this chapter, we have seen methods such as `.map`, `.filter`, and `.sum`, which implement certain kinds of iterative computations. These and other operations can be combined in very flexible ways, which allows programmers to write iterative code *without loops*.

The programmer can avoid writing loops because the iteration is delegated to the library functions `.map`, `.filter`, `.sum`, and so on. It is the job of the library and the compiler to translate these functions into machine code. The machine code most likely *will* contain loops; but the functional programmer does not need to see that code or to reason about it.

1.7.5 Nameless functions in mathematical notation

Functions in mathematics are mappings from one set to another. A function does not necessarily *need* a name; the mapping just needs to be defined. However, nameless functions have not been widely used in the conventional mathematical notation. It turns out that nameless functions are quite important in functional programming because, in particular, they allow programmers to write code more concisely and use a straightforward, consistent syntax.

Nameless functions have the property that their bound variables are invisible outside their scope. This property is directly reflected by the prevailing mathematical conventions. Compare the formulas

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ; \quad f(x) = \int_0^x \frac{dz}{1+z} \quad .$$

The mathematical convention is that these formulas define the same function f, and that one may rename the integration variable at will.

In programming, the only situation when a variable "may be renamed at will" is when the variable represents an argument of a function. It follows that the notations $\frac{dx}{1+x}$ and $\frac{dz}{1+z}$ correspond to a nameless function whose argument was renamed from x to z. In FP notation, this nameless function would be denoted as $z \Rightarrow \frac{1}{1+z}$, and the integral rewritten as code such as

$$\text{integration}\,(0, x, g) \;\text{ where } g = \left(z \Rightarrow \frac{1}{1+z}\right) \quad .$$

Now consider the traditional mathematical notations for summation, e.g.

$$\sum_{k=0}^{x} \frac{1}{1+k} \quad .$$

In sums, the bound variable k is introduced under the $\sum$ symbol; but in integrals, the bound variable follows the special symbol "d". This notational inconsistency could be removed if we were to use nameless functions explicitly, for example:

$$\sum_{0}^{x} \left(k \Rightarrow \frac{1}{1+k} \right) \text{ instead of } \sum_{k=0}^{x} \frac{1}{1+k} \quad ,$$
$$\int_{0}^{x} \left(z \Rightarrow \frac{1}{1+z} \right) \text{ instead of } \int_{0}^{x} \frac{dz}{1+z} \quad .$$

In this notation, the new summation symbol $\sum_0^x$ does not mention the name "k" but takes a function as an argument. Similarly, the new integration symbol $\int_0^x$ does not mention "z" and does not use the special symbol "d" but now takes a function as an argument. Written in this way, the operations of summation and integration become *functions* that take a function as argument. The above summation may be written in a consistent and straightforward manner as a function:

$$\text{summation}\,(0, x, f) \text{ where } f = \left(y \Rightarrow \frac{1}{1+y} \right) \quad .$$

We could implement `summation(a,b,g)` as

```
def summation(a: Int, b: Int, g: Int => Double) = (a to b).map(g).sum

scala> summation(1, 10, x => math.sqrt(x))
res0: Double = 22.4682781862041
```

Numerical integration requires longer code, since the formulas are more complicated. For instance, Simpson's rule can be written as

$$\text{integration}\,(a, b, g) = \frac{\delta}{3}\left(g(a) + g(b) + 4s_1 + 2s_2\right) \quad ,$$
$$\text{where} \quad n = 2\left\lfloor \frac{b-a}{\varepsilon} \right\rfloor, \quad \delta_x = \frac{b-a}{n} \quad ,$$
$$s_1 = \sum_{i=1,3,\ldots,n-1} g(a + i\delta_x) \quad ,$$
$$s_2 = \sum_{i=2,4,\ldots,n-2} g(a + i\delta_x) \quad .$$

A straightforward translation of this formula into Scala is

```
def integration(a: Double, b: Double, g: Double => Double, eps: Double) = {
  // First, we define some helper values and functions that replace
  // the definitions "where n = ..." in the mathematical formula.
   val n: Int = (math.round((b-a)/eps/2)*2).toInt
   val delta_x = (b - a) / n
   val g_i = (i: Int) => g(a + i*delta_x)
   val s1 = (1 to (n-1) by 2).map(g_i).sum
   val s2 = (2 to (n-2) by 2).map(g_i).sum
   // Now we write the expression for the final result.
   delta_x / 3 * (g(a) + g(b) + 4*s1 + 2*s2)
}
```

```
scala> integration(0, 2, x => x*x*x, 0.001) // Exact answer is 4
res0: Double = 4.000000000000003

scala> integration(0, 7, x => x*x*x*x*x*x*x, 0.001) // Exact answer is 117649
res1: Double = 117649.00000000023
```

The entire code is one large *expression*, with a few sub-expressions defined for convenience as a few helper values and helper functions. In other words, this code is written in the FP paradigm.

1.7.6 Named and nameless expressions and their uses

It is a significant advantage if a programming language supports unnamed (or "nameless") expressions. To see this, consider a familiar situation where we take the absence of names for granted.

In most programming languages today, we can directly write arithmetical expressions such as `(x+123)*y/(2+x)`. Here, `x` and `y` are variables with names. Note, however, that the entire expression does not need to have a name. Parts of that expression (such as `x+123` or `2+x`) also do not need to have separate names. It would be quite inconvenient if we *needed* to assign a name separately to each sub-expression. The code for `(x+123)*y/(2+x)` could then look like this:

```
r1 = 123
r2 = x + r1
r3 = r2 * y
r4 = 2
r5 = r4 + x
r6 = r3 / r5
return r6
```

This style of programming resembles assembly languages, where *every* sub-expression – that is, every step of every calculation, – must be named separately (and, in the assembly languages, assigned a memory address or a CPU register).

So, programmers become more productive when their programming language supports nameless expressions.

This is also common practice in mathematics; names are assigned when needed, but most expressions remain nameless.

It is similarly quite useful if data structures can be declared without a name. For instance, a **dictionary** (also called a "hashmap") is declared in Scala as

```
Map("a" -> 1, "b" -> 2, "c" -> 3)
```

This is a nameless expression representing a dictionary. Without this construction, programmers have to write cumbersome, repetitive code that creates an initially empty dictionary and then fills it step by step with values:

```
// Scala code for creating a dictionary:
val myMap = Map("a" -> 1, "b" -> 2, "c" -> 3)
// Java code:
// Map<String, Int> myMap = new HashMap<String, Integer>() {{
//   put("a", 1);
//   put("b", 2);
//   put("c", 3);
// }}; // The shortest Java code for creating the same dictionary.
```

Nameless functions are useful for the same reason as nameless data values: they allow us to build larger programs from simpler parts in a uniform way.

1.7.7 Nameless functions: historical perspective

Nameless functions were first used in 1936 in a theoretical programming language called "λ-calculus". In that language,[4] all functions are nameless and have a single argument. The letter λ is a syntax separator denoting function arguments in nameless functions. For example, the nameless function $x \Rightarrow x + 1$ could be written as $\lambda x.add\ x\ 1$ in λ-calculus, if it had a function add for adding integers (which it does not).

In most programming languages that were in use until around 1990, all functions required names. But by 2015, most languages added support for nameless functions, because programming in the map/reduce style (which invites frequent use of nameless functions) turned out to be immensely useful. Table 1.2 shows the year when nameless functions were introduced in each language.

What this book calls a "nameless function" is also called anonymous function, function expression, function literal, closure, lambda function, lambda expression, or just a "lambda". I use the term "nameless function" in this book because it is the most descriptive and unambiguous both in speech and in writing.

[4] Although called a "calculus," it is in reality a (drastically simplified) programming language. It has nothing to do with "calculus" as known in mathematics, such as differential or integral calculus. Also, the letter λ has no particular significance; it plays a purely syntactic role in the λ-calculus. Practitioners of functional programming usually do not need to study any λ-calculus. All practically relevant knowledge related to λ-calculus is explained in Chapter 4 of this book.

Language	Year	Code for $k^{:\mathrm{Int}} \Rightarrow k + k$
λ-calculus	1936	$\lambda k.\ add\ k\ k$
typed λ-calculus	1940	$\lambda k : int.\ add\ k\ k$
LISP	1958	`(lambda (k) (+ k k))`
Standard ML	1973	`fn (k:int) => k + k`
Scheme	1975	`(lambda (k) (+ k k))`
OCaml	1985	`fun (k:int) -> k + k`
Haskell	1990	`\ k -> (k::Int) + k`
Oz	1991	`fun {$ K} K + K`
R	1993	`function(k) k + k`
Python 1.0	1994	`lambda k:  k + k`
JavaScript	1995	`function(k) { return k + k; }`
Mercury	1995	`func(K) = K + K`
Ruby	1995	`lambda { |k| k + k }`
Lua 3.1	1998	`function(k) return k + k end`
Scala	2003	`(k:Int) => k + k`
F#	2005	`fun (k:int) -> k + k`
C# 3.0	2007	`delegate(int k) { return k + k; }`
C++ 11	2011	`[] (int k) { return k + k; }`
Go	2012	`func(k int) { return k + k }`
Kotlin	2012	`{ k:Int -> k + k }`
Swift	2014	`{ (k:int) -> int in return k + k }`
Java 8	2014	`(int k) -> k + k`
Rust	2015	`|k:i32| k + k`

Table 1.2: Nameless functions in various programming languages.

2 Mathematical formulas as code. II. Mathematical induction

We will now study more flexible ways of working with data collections in the functional programming paradigm. The Scala standard library has methods for performing general iterative computations, that is, computations defined by induction. Translating mathematical induction into code is the focus of this chapter.

But first we need to become fluent in using tuple types with Scala collections.

2.1 Tuple types

2.1.1 Examples of using tuples

Many standard library methods in Scala require working with **tuple** types. A simple example of a tuple is a *pair* of values, – such as, a pair of an integer and a string. The Scala syntax for this type of pair is

```
val a: (Int, String) = (123, "xyz")
```

The type expression `(Int, String)` denotes this tuple type.

A **triple** is defined in Scala like this:

```
val b: (Boolean, Int, Int) = (true, 3, 4)
```

Pairs and triples are examples of tuples. A **tuple** can contain any number of values, which I call **parts** of a tuple. The parts of a tuple can have different types, but the type of each part is fixed once and for all. Also, the number of parts in a tuple is fixed. It is a **type error** to use incorrect types in a tuple, or an incorrect number of parts of a tuple:

```
scala> val bad: (Int, String) = (1,2)
<console>:11: error: type mismatch;
 found   : Int(2)
 required: String
       val bad: (Int, String) = (1,2)
                                 ^

scala> val bad: (Int, String) = (1,"a",3)
<console>:11: error: type mismatch;
 found   : (Int, String, Int)
 required: (Int, String)
       val bad: (Int, String) = (1,"a",3)
                                ^
```

Parts of a tuple can be accessed by number, starting from 1. The Scala syntax for **tuple accessor** methods looks like `._1`, for example:

```
scala> val a = (123, "xyz")
a: (Int, String) = (123,xyz)

scala> a._1
res0: Int = 123
```

```
scala> a._2
res1: String = xyz
```

It is a type error to access a tuple part that does not exist:

```
scala> a._0
<console>:13: error: value _0 is not a member of (Int, String)
       a._0
         ^

scala> a._5
<console>:13: error: value _5 is not a member of (Int, String)
       a._5
         ^
```

Type errors are detected at compile time, before any computations begin.

Tuples can be **nested**: any part of a tuple can be itself of a tuple type.

```
scala> val c: (Boolean, (String, Int), Boolean) = (true, ("abc", 3), false)
c: (Boolean, (String, Int), Boolean) = (true,(abc,3),false)

scala> c._1
res0: Boolean = true

scala> c._2
res1: (String, Int) = (abc,3)
```

To define functions whose arguments are tuples, we could use the tuple accessors. An example of such a function is

```
def f(p: (Boolean, Int), q: Int): Boolean = p._1 && (p._2 > q)
```

The first argument, `p`, of this function, has a tuple type. The function body uses accessor methods (`._1` and `._2`) to compute the result value. Note that the second part of the tuple `p` is of type `Int`, so it is valid to compare it with an integer `q`. It would be a type error to compare the *tuple* `p` with an *integer* using the expression `p > q`. It would be also a type error to apply the function `f` to an argument `p` that has a wrong type, e.g. the type `(Int, Int)` instead of `(Boolean, Int)`.

2.1.2 Pattern matching for tuples

Instead of using accessor methods when working with tuples, it is often convenient to use **pattern matching**. Pattern matching occurs in two situations in Scala:

- destructuring definition: `val` *pattern* `= ...`
- `case` expression: `case` *pattern* `=> ...`

An example of a **destructuring definition** is

```
scala> val g = (1, 2, 3)
g: (Int, Int, Int) = (1,2,3)

scala> val (x, y, z) = g
x: Int = 1
y: Int = 2
z: Int = 3
```

The value `g` is a tuple of three integers. After defining `g`, we define the three variables `x`, `y`, `z` *at once* in a single `val` definition. We imagine that this definition "destructures" the data structure contained

in `g` and decomposes it into three parts, then assigns the names `x`, `y`, `z` to these parts. The types of the new values are also assigned automatically.

The left-hand side of the destructuring definition contains the tuple pattern `(x, y, z)` that looks like a tuple, except that its parts are names `x`, `y`, `z` that are so far *undefined*. These names are called **pattern variables**. The destructuring definition checks whether the structure of the value of `g` "matches" the three pattern variables. (If `g` does not contain a tuple with exactly three parts, the definition will fail.) This computation is called **pattern matching**.

Pattern matching is often used when working with tuples:

```
scala> (1, 2, 3) match { case (a, b, c) => a + b + c }
res0: Int = 6
```

The **case expression** (`case (a, b, c) => ...`) performs pattern matching on the tuple argument `p`. The pattern matching will "destructure" (i.e. decompose) the tuple and try to match it to the given pattern `(a, b, c)`. In this pattern, `a`, `b`, `c` are as yet undefined new variables, – that is, they are pattern variables. If the pattern matching succeeds, the pattern variables `a`, `b`, `c` are assigned their values, and the function body can proceed to perform its computation. In this example, the pattern variables `a`, `b`, `c` will be assigned values 1, 2, and 3, so the function returns 6 as its result value.

Pattern matching is especially convenient when working with nested tuples. Here is an example of such code:

```
def t1(p: (Int, (String, Int))): String = p match {
  case (x, (str, y)) => str + (x + y).toString
}

scala> t((10, ("result is ", 2)))
res0: String = result is 12
```

The type structure of the argument is visually repeated in the pattern. It is easy to see that `x` and `y` become integers and `str` becomes a string after pattern matching. If we rewrite the same code using the tuple accessor methods instead of pattern matching, the code will look like this:

```
def t2(p: (Int, (String, Int))): String = p._2._1 + (p._1 + p._2._2).toString
```

This code is shorter but harder to read: For example, it is not immediately clear what `p._2._1` refers to. It is also harder to change this code: Suppose we want to change the type of the tuple `p` to `((Int, String), Int)`. Then the new code is

```
def t3(p: ((Int, String), Int)): String = p._1._2 + (p._1._1 + p._2).toString
```

It takes time to verify, by going through every accessor method, that the function `t3` computes the same expression as `t2`. In contrast, the code is changed easily when using the pattern matching expression instead of the accessor methods:

```
def t4(p: ((Int, String), Int)): String = p match {
  case ((x, str), y) => str + (x + y).toString
}
```

The only change in the function body, compared to `t1`, is in the pattern matcher, so it is visually clear that `t4` computes the same expression as `t1`.

Sometimes we do not need some of the tuple parts in a pattern match. The following syntax is used to make this intention clear:

```
scala> val (x, _, _, z) = ("abc", 123, false, true)
x: String = abc
z: Boolean = true
```

The underscore symbol `_` denotes the parts of the pattern that we want to ignore. The underscore will always match any value regardless of type.

A feature of Scala is a short syntax for functions such as `{case (x, y) => y}` that extract elements from tuples. The shorter syntax is `(t => t._2)` or even shorter, `(_._2)`, as illustrated here:

```
scala> val p: ((Int, Int )) => Int = { case (x, y) => y }
p: ((Int, Int)) => Int = <function1>

scala> p((1, 2))
res0: Int = 2

scala> val q: ((Int, Int )) => Int = (_._2)
q: ((Int, Int)) => Int = <function1>

scala> q((1, 2))
res1: Int = 2

scala> Seq( (1,10), (2,20), (3,30) ).map(_._2)
res2: Seq[Int] = List(10, 20, 30)
```

2.1.3 Using tuples with collections

Tuples can be combined with any other types without restrictions. For instance, we can define a tuple of functions,

```
val q: (Int => Int, Int => Int) = (x => x + 1, x => x - 1)
```

We can create a list of tuples,

```
val r: List[(String, Int)] = List(("apples", 3), ("oranges", 2), ("pears", 0))
```

We could define a tuple of lists of tuples of functions, or any other combination.

Here is an example of using the standard method `.map` to transform a list of tuples. The argument of `.map` must be a function taking a tuple as its argument. It is convenient to use pattern matching for writing such functions:

```
scala> val basket: List[(String, Int)] = List(("apples", 3), ("pears", 2), ("lemons", 0))
basket: List[(String, Int)] = List((apples,3), (pears,2), (lemons,0))

scala> basket.map { case (fruit, count) => count * 2 }
res0: List[Int] = List(6, 4, 0)

scala> basket.map { case (fruit, count) => count * 2 }.sum
res1: Int = 10
```

In this way, we can use the standard methods such as `.map`, `.filter`, `.max`, `.sum` to manipulate sequences of tuples. The names "`fruit`", "`count`" are chosen to help us remember the meaning of the parts of tuples.

We can easily transform a list of tuples into a list of values of a different type:

```
scala> basket.map { case (fruit, count) =>
  val isAcidic = fruit == "lemons"
  (fruit, isAcidic)
}
res2: List[(String, Boolean)] = List((apples,false), (pears,false), (lemons,true))
```

In the Scala syntax, a nameless function written with braces `{ ... }` can define local values in its body. The return value of the function is the last expression written in the function body. In this example, the return value of the nameless function is the tuple `(fruit, isAcidic)`.

2.1.4 Treating dictionaries (Scala's `Map`s) as collections

In the Scala standard library, tuples are frequently used as types of intermediate values. For instance, tuples are used when iterating over dictionaries. The Scala type `Map[K,V]` represents a dictionary with keys of type `K` and values of type `V`. Here `K` and `V` are **type parameters**. Type parameters represent unknown types that will be chosen later, when working with values having specific types.

In order to create a dictionary with given keys and values, we can write

```
Map(("apples", 3), ("oranges", 2), ("pears", 0))
```

This is equivalent to first creating a sequence of key/value *pairs* and then converting that sequence into a dictionary.

Pairs are used often, so the Scala library defines a special infix syntax for pairs via the arrow symbol `->`. The expression `x -> y` is equivalent to the pair `(x, y)`:

```
scala> "apples" -> 3
res0: (String, Int) = (apples,3)
```

With this syntax, it is easier to read the code for creating a dictionary:

```
Map("apples" -> 3, "oranges" -> 2, "pears" -> 0)
```

A list of pairs can be converted to a dictionary using the method `.toMap`. The same method works for other collection types such as `Seq`, `Vector`, `Stream`, and `Array`.

The method `.toSeq` converts a dictionary into a sequence of pairs:

```
scala> Map("apples" -> 3, "oranges" -> 2, "pears" -> 0).toSeq
res20: Seq[(String, Int)] = ArrayBuffer((apples,3), (oranges,2), (pears,0))
```

The `ArrayBuffer` is one of the many list-like data structures in the Scala library. All these data structures are gathered under the common "sequence" type called `Seq`. The methods defined in the Scala standard library sometimes return different implementations of the `Seq` type for reasons of performance.

The standard library has several useful methods that use tuple types, such as `.map` and `.filter` (with dictionaries), `.toMap`, `.zip`, and `.zipWithIndex`. The methods `.flatten`, `.flatMap`, `.groupBy`, and `.sliding` also work with most collection types, including dictionaries and sets. It is important to become familiar with these methods, because it will help writing code that uses sequences, sets, and dictionaries. Let us now look at these methods one by one.

The `.map` and `.toMap` methods Chapter 1 showed how the `.map` method works on sequences: the expression `xs.map(f)` applies a given function `f` to each element of the sequence `xs`, gathering the results in a new sequence. In this sense, we can say that the `.map` method "iterates over" sequences. The `.map` method works similarly on dictionaries, except that iterating over a dictionary of type `Map[K, V]` when applying `.map` looks like iterating over a sequence of *pairs*, `Seq[(K,V)]`. If `d:Map[K,V]` is a dictionary, the argument `f` of `d.map(f)` must be a function operating on tuples of type `(K,V)`. Typically, such functions are written using `case` expressions:

```
val m1 = Map("apples" -> 3, "pears" -> 2, "lemons" -> 0)

scala> m1.map { case (fruit, count) => count * 2 }
res0: Seq[Int] = ArrayBuffer(6, 4, 0)
```

If we want to transform a dictionary into another dictionary, we can first create a sequence of pairs and then convert it to a dictionary with the `.toMap` method:

```
scala> m1.map { case (fruit, count) => (fruit, count * 2) }.toMap
res1: Map[String,Int] = Map(apples -> 6, pears -> 4, lemons -> 0)
```

The `.filter` method works on dictionaries by iterating on key/value pairs. The filtering predicate must be a function of type `((K, V)) => Boolean`. For example:

```
scala> m1.filter { case (fruit, count) => count > 0 }.toMap
res2: Map[String,Int] = Map(apples -> 6, pears -> 4)
```

The `.zip` and `.zipWithIndex` methods The `.zip` method takes *two* sequences and produces a sequence of pairs, taking one element from each sequence:

```
scala> val s = List(1, 2, 3)
s: List[Int] = List(1, 2, 3)

scala> val t = List(true, false, true)
t: List[Boolean] = List(true, false, true)

scala> s.zip(t)
res3: List[(Int, Boolean)] = List((1,true), (2,false), (3,true))

scala> s zip t
res4: List[(Int, Boolean)] = List((1,true), (2,false), (3,true))
```

In the last line, the equivalent "dotless" infix syntax (`s zip t`) is shown just to illustrate the flexibility of syntax conventions in Scala.

The `.zip` method works equally well on dictionaries: in that case, dictionaries are automatically converted to sequences of tuples before applying `.zip`.

The `.zipWithIndex` method transforms a sequence into a sequence of pairs, where the second part of the pair is the zero-based index:

```
scala> List("a", "b", "c").zipWithIndex
res5: List[(String, Int)] = List((a,0), (b,1), (c,2))
```

The `.flatten` method converts nested sequences to "flattened" ones:

```
scala> List(List(1, 2), List(2, 3), List(3, 4)).flatten
res6: List[Int] = List(1, 2, 2, 3, 3, 4)
```

The "flattening" operation computes the concatenation of all inner sequences. In Scala, sequences are concatenated using the operation `++`, e.g.:

```
scala> List(1, 2, 3) ++ List(4, 5, 6) ++ List(0)
res7: List[Int] = List(1, 2, 3, 4, 5, 6, 0)
```

So the `.flatten` method inserts the operation `++` between all the inner sequences.

Keep in mind that `.flatten` removes *only one* level of nesting, which is at the "outside" of the data structure. If applied to a `List[List[List[Int]]]`, the `.flatten` method returns a `List[List[Int]]`:

```
scala> List(List(List(1), List(2)), List(List(2), List(3))).flatten
res8: List[List[Int]] = List(List(1), List(2), List(2), List(3))
```

The `.flatMap` method is closely related to `.flatten` and can be seen as a shortcut, equivalent to first applying `.map` and then `.flatten`:

```
scala> List(1,2,3,4).map(n => (1 to n).toList)
res9: List[List[Int]] = List(List(1), List(1, 2), List(1, 2, 3), List(1, 2, 3, 4))

scala> List(1,2,3,4).map(n => (1 to n).toList).flatten
res10: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)

scala> List(1,2,3,4).flatMap(n => (1 to n).toList)
res11: List[Int] = List(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

The `.flatMap` operation transforms a sequence by mapping each element to a potentially different number of new elements.

At first sight, it may be unclear why `.flatMap` is useful. (Should we perhaps combine `.filter` and `.flatten` into a `.flatFilter`, or combine `.zip` and `.flatten` into a `.flatZip`?) However, we will see later in this book that the use of `.flatMap`, which is related to "monads", is one of the most versatile and powerful design patterns in functional programming. In this chapter, several examples and exercises will illustrate the use of `.flatMap` for working on sequences.

The `.groupBy` method rearranges a sequence into a dictionary where some elements of the original sequence are grouped together into subsequences. For example, given a sequence of words, we can group all words that start with the letter `"y"` into one subsequence, and all other words into another subsequence. This is accomplished by the following code,

```
scala> Seq("wombat", "xanthan", "yoghurt", "zebra").
  groupBy(s => if (s startsWith "y") 1 else 2)
res12: Map[Int,Seq[String]] = Map(1 -> List(yoghurt), 2 -> List(wombat, xanthan, zebra))
```

The argument of the `.groupBy` method is a *function* that computes a "key" out of each sequence element. The key can have an arbitrarily chosen type. (In the current example, that type is `Int`.) The result of `.groupBy` is a dictionary that maps each key to the sub-sequence of values that have that key. (In the current example, the type of the dictionary is therefore `Map[Int, Seq[String]]`.) The order of elements in the sub-sequences remains the same as in the original sequence.

As another example of using `.groupBy`, the following code will group together all numbers that have the same remainder after division by 3:

```
scala> List(1,2,3,4,5).groupBy(k => k % 3)
res13: Map[Int,List[Int]] = Map(2 -> List(2, 5), 1 -> List(1, 4), 0 -> List(3))
```

The `.sliding` method creates a sliding window of a given width and returns a sequence of nested sequences:

```
scala> (1 to 10).sliding(4).toList
res14: List[IndexedSeq[Int]] = List(Vector(1, 2, 3, 4), Vector(2, 3, 4, 5), Vector(3, 4, 5, 6),
    Vector(4, 5, 6, 7), Vector(5, 6, 7, 8), Vector(6, 7, 8, 9), Vector(7, 8, 9, 10))
```

Usually, this method is used together with an aggregation operation on the inner sequences. For example, the following code computes a sliding-window average with window width 50 over an array of 100 numbers:

```
scala> (1 to 100).map(x => math.cos(x)).sliding(50).
  map(_.sum / 50).take(5).toList
res15: List[Double] = List(-0.005153079196990285, -0.0011160413780774369, 0.003947079736951305,
    0.005381273944717851, 0.0018679497047270743)
```

The `.sortBy` method sorts a sequence according to a sorting key. The argument of `.sortBy` is a *function* that computes the sorting key from a sequence element. In this way, we can sort elements in an arbitrary way:

```
scala> Seq(1, 2, 3).sortBy(x => -x)
res0: Seq[Int] = List(3, 2, 1)

scala> Seq("z", "xxx", "yy").sortBy(word => word)
res1: Seq[String] = List("xxx", "yy", "z")

scala> Seq("z", "xxx", "yy").sortBy(word => word.length)
res2: Seq[String] = List("z", "yy", "xxx")
```

Sorting by the elements themselves, as we have done here with `.sortBy(word => word)`, is only possible if the element's type has a well-defined ordering. For strings, this is the alphabetic ordering, and for integers, the standard arithmetic ordering. For such types, a convenience method `.sorted` is defined, and works equivalently to `.sortBy(x => x)`:

```
scala> Seq("z", "xxx", "yy").sorted
res3: Seq[String] = List("xxx", "yy", "z")
```

2.1.5 Solved examples: Tuples and collections

Example 2.1.5.1 For a given sequence x_i, compute the sequence of pairs $b_i = (\cos x_i, \sin x_i)$.

Hint: use `.map`, assume `xs:Seq[Double]`.

Solution: We need to produce a sequence that has a pair of values corresponding to each element of the original sequence. This transformation is exactly what the `.map` method does. So the code is

```
xs.map { x => (math.cos(x), math.sin(x)) }
```

Example 2.1.5.2 Count how many times $\cos x_i > \sin x_i$ occurs in a sequence x_i.

Hint: use `.count`, assume `xs:Seq[Double]`.

Solution: The method `.count` takes a predicate and returns the number of times the predicate was `true` while evaluated on the elements of the sequence:

```
xs.count { x => math.cos(x) > math.sin(x) }
```

We could also reuse the solution of Exercise 2.1.5.1 that computed the cosine and the sine values. The code would then become

```
xs.map { x => (math.cos(x), math.sin(x)) }
  .count { case (cosine, sine) => cosine > sine }
```

Example 2.1.5.3 For given sequences a_i and b_i, compute the sequence of differences $c_i = a_i - b_i$.

Hint: use `.zip`, `.map`, and assume `as` and `bs` are of type `Seq[Double]`.

Solution: We can use `.zip` on `as` and `bs`, which gives a sequence of pairs,

```
as.zip(bs) : Seq[(Double, Double)]
```

We then compute the differences $a_i - b_i$ by applying `.map` to this sequence:

```
as.zip(bs).map { case (a, b) => a - b }
```

Example 2.1.5.4 In a given sequence p_i, count how many times $p_i > p_{i+1}$ occurs.

Hint: use `.zip` and `.tail`.

Solution: Given `ps:Seq[Double]`, we can compute `ps.tail`. The result is a sequence that is 1 element shorter than `ps`, for example:

```
scala> val ps = Seq(1,2,3,4)
ps: Seq[Int] = List(1, 2, 3, 4)

scala> ps.tail
res0: Seq[Int] = List(2, 3, 4)
```

Taking a `.zip` of the two sequences `ps` and `ps.tail`, we get a sequence of pairs:

```
scala> ps.zip(ps.tail)
res1: Seq[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Note that `ps.tail` is 1 element shorter than `ps`, and the resulting sequence of pairs is also 1 element shorter than `ps`. In other words, it is not necessary to truncate `ps` before computing `ps.zip(ps.tail)`. Now apply the `.count` method:

```
ps.zip(ps.tail).count { case (a, b) => a > b }
```

Example 2.1.5.5 For a given $k > 0$, compute the sequence $c_i = \max(b_{i-k}, ..., b_{i+k})$.

Hint: use `.sliding`.

Solution: Applying the `.sliding` method to a list gives a list of nested lists:

```
scala> val bs = List(1, 2, 3, 4, 5)
bs: List[Int] = List(1, 2, 3, 4, 5)

scala> bs.sliding(3).toList
res0: List[List[Int]] = List(List(1, 2, 3), List(2, 3, 4), List(3, 4, 5))
```

For each b_i, we need to obtain a list of $2k + 1$ nearby elements $(b_{i-k}, ..., b_{i+k})$. So we need to use `.sliding(2*k+1)` to obtain a window of the required size. Now we can compute the maximum of each of the nested lists by using the `.map` method on the outer list, with the `.max` method applied to the nested lists. So the argument of the `.map` method must be the function `nested => nested.max`. The final code is

```
bs.sliding(2 * k + 1).map(nested => nested.max)
```

In Scala, this code can be written more concisely using the syntax

```
bs.sliding(2 * k + 1).map(_.max)
```

because `_.max` means the nameless function `x => x.max`.

Example 2.1.5.6 Create a 10×10 multiplication table as a dictionary of type `Map[(Int, Int), Int]`. For example, a 3×3 multiplication table would be given by this dictionary,

```
Map( (1, 1) -> 1, (1, 2) -> 2, (1, 3) -> 3, (2, 1) -> 2,
  (2, 2) -> 4, (2, 3) -> 6, (3, 1) -> 3, (3, 2) -> 6, (3, 3) -> 9 )
```

Hint: use `.flatMap` and `.toMap`.

Solution: We are required to make a dictionary that maps pairs of integers `(x, y)` to `x * y`. Begin by creating the list of *keys* for that dictionary, which must be a list of pairs `(x, y)` of the form `List((1,1), (1,2), ..., (2,1), (2,2), ...)`. We need to start with a sequence of values of `x`, and for each `x` from that sequence, iterate over another sequence to provide values for `y`. Try this computation:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3))
s: List[List[Int]] = List(List(1, 2, 3), List(1, 2, 3), List(1, 2, 3))
```

We would like to get `List((1,1), (1,2), 1,3))` etc., and so we use `.map` on the inner list with a nameless function `y => (x, y)` that converts a number into a tuple,

```
scala> List(1, 2, 3).map{ y => (x, y) }
res0: List[(Int, Int)] = List((1,1), (1,2), (1,3))
```

Here the curly braces `{ y => (x, y) }` are used only for clarity; we could equivalently use parentheses and write `(y => (x, y))`.

Using this `.map` operation, we obtain the code for a nested list of tuples:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3).map{ y => (x, y) })
s: List[List[(Int, Int)]] = List(List((1,1), (1,2), (1,3)), List((2,1), (2,2), (2,3)), List((3,1),
    (3,2), (3,3)))
```

This is almost what we need, except that the nested lists need to be concatenated into a single list. This is exactly what `.flatten` does:

```
scala> val s = List(1, 2, 3).map(x => List(1, 2, 3).map{ y => (x, y) }).flatten
s: List[(Int, Int)] = List((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

It is shorter to write `.flatMap(...)` instead of `.map(...).flatten`:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).map{ y => (x, y) })
s: List[(Int, Int)] = List((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

This is the list of keys for the required dictionary, which should map each *pair* of integers `(x,y)` to `x*y`. In Scala, a dictionary is usually created by applying `.toMap` to a sequence of pairs `(key,value)`. So we need to create a sequence of nested tuples of the form `((x,y), product)`. To achieve this, we use `.map` with a function that computes the product and creates a nested tuple:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).
  map{ y => (x, y) }).map{ case (x, y) => ((x, y), x * y) }
s: List[((Int, Int), Int)] = List(((1,1),1), ((1,2),2), ((1,3),3), ((2,1),2), ((2,2),4), ((2,3),6),
     ((3,1),3), ((3,2),6), ((3,3),9))
```

We can simplify this code if we notice that we are first mapping each `y` to a tuple `(x, y)`, and later map each tuple `(x,y)` to a nested tuple `((x,y), x*y)`. Instead, the entire computation can be done in the inner `.map` operation:

```
scala> val s = List(1, 2, 3).flatMap(x => List(1, 2, 3).
  map{ y => ((x,y), x*y) })
s: List[((Int, Int), Int)] = List(((1,1),1), ((1,2),2), ((1,3),3), ((2,1),2), ((2,2),4), ((2,3),6),
     ((3,1),3), ((3,2),6), ((3,3),9))
```

It remains to convert this list of tuples to a dictionary with `.toMap`. Also, for better readability, we can use Scala's pair syntax `key -> value`, which is completely equivalent to writing the tuple `(key,value)`. The final code is

```
(1 to 10).flatMap(x => (1 to 10).map{ y => (x,y) -> x*y }).toMap
```

Example 2.1.5.7 For a given sequence x_i, compute the maximum of all of the numbers x_i, $\cos x_i$, $\sin x_i$. Hint: use `.flatMap`, `.max`.

Solution: We will compute the required value if we take `.max` of a list containing all of the numbers. To do that, first map each element of the list `xs:Seq[Double]` into a sequence of three numbers:

```
scala> List(0.1, 0.5, 0.9)
res0: List[Double] = List(0.1, 0.5, 0.9)

scala> res0.map{ x => Seq(x, math.cos(x), math.sin(x)) }
res1: List[Seq[Double]] = List(List(0.1, 0.9950041652780258, 0.09983341664682815), List(0.5,
     0.8775825618903728, 0.479425538604203), List(0.9, 0.6216099682706644, 0.7833269096274834))
```

This list is almost what we need, except we need to `.flatten` it:

```
scala> res1.flatten
res2: List[Double] = List(0.1, 0.9950041652780258, 0.09983341664682815, 0.5, 0.8775825618903728,
     0.479425538604203, 0.9, 0.6216099682706644, 0.7833269096274834)
```

Now we just need to take the maximum of the resulting numbers:

```
scala> res2.max
res3: Double = 0.9950041652780258
```

The final code (starting from a given sequence `xs`) is

```
xs.flatMap{ x => Seq(x, math.cos(x), math.sin(x)) }.max
```

Example 2.1.5.8 From a dictionary of type `Map[String, String]` mapping names to addresses, and assuming that the addresses do not repeat, compute a dictionary of type `Map[String, String]` mapping the addresses back to names.

Hint: use `.map` and `.toMap`.

Solution: Keep in mind that iterating over a dictionary looks like iterating over a list of `(key, value)` pairs, and use `.map` to reverse each pair:

```
dict.map{ case (name, addr) => (addr, name) }.toMap
```

Example 2.1.5.9 Write the solution of Example 2.1.5.8 as a function with type parameters `Name` and `Addr` instead of the fixed type `String`.

Solution: In Scala, the syntax for type parameters in a function definition is

```
def rev[Name, Addr](...) = ...
```

The type of the argument is `Map[Name, Addr]`, while the type of the result is `Map[Addr, Name]`. So we use the type parameters `Name` and `Addr` in the type signature of the function. The final code is

```
def rev[Name, Addr](dict: Map[Name, Addr]): Map[Addr, Name] =
  dict.map { case (name, addr) => (addr, name) }.toMap
```

The body of the function `rev` remains the same as in Example 2.1.5.8; only the type signature changed. This is because the procedure for reversing a dictionary works in the same way for dictionaries of any type. So the body of the function `rev` does not actually need to know the types of the keys and values in the dictionary. For this reason, it was easy for us to change the specific types (`String`) into type parameters in this function.

When the function `rev` is applied to a dictionary of a specific type, the Scala compiler will automatically set the type parameters `Name` and `Addr` that fit the required types of the dictionary's keys and values. For example, if we apply `rev` to a dictionary of type `Map[Boolean, Seq[String]]`, the type parameters will be set automatically as `Name = Boolean` and `Addr = Seq[String]`:

```
scala> val d = Map(true -> Seq("x", "y"), false -> Seq("z", "t"))
d: Map[Boolean, Seq[String]] = Map(true -> List(x, y), false -> List(z, t))

scala> rev(d)
res0: Map[Seq[String], Boolean] = Map(List(x, y) -> true, List(z, t) -> false)
```

Type parameters can be also set explicitly when using the function `rev`. If the type parameters are chosen incorrectly, the program will not compile:

```
scala> rev[Boolean, Seq[String]](d)
res1: Map[Seq[String],Boolean] = Map(List(x, y) -> true, List(z, t) -> false)

scala> rev[Int, Double](d)
<console>:14: error: type mismatch;
 found   : Map[Boolean,Seq[String]]
 required: Map[Int,Double]
       rev[Int, Double](d)
                        ^
```

Example 2.1.5.10* Given a sequence `words:Seq[String]` of words, compute a sequence of type `Seq[(Seq[String], Int)]`, where each inner sequence contains all the words having the same length, paired with the integer value showing that length. So, the input `Seq("the", "food", "is", "good")`

should produce the output

```
Seq( (Seq("is"), 2), (Seq("the"), 3), (Seq("food", "good"), 4) )
```

The resulting sequence must be ordered by increasing length of words.

Solution: It is clear that we need to begin by grouping the words by length. The library method `.groupBy` takes a function that computes a grouping key from each element of a sequence. In our case, we need to group by word length, which is computed with the method `.length` if applied to a string. So the first step is

```
words.groupBy{ word => word.length }
```

or, more concisely, `words.groupBy(_.length)`. The result of this expression is a dictionary that maps each length to the list of words having that length:

```
scala> words.groupBy(_.length)
res0: scala.collection.immutable.Map[Int,Seq[String]] = Map(2 -> List(is), 4 -> List(food, good), 3
     -> List(the))
```

This is already close to what we need. If we convert this dictionary to a sequence, we will get a list of pairs

```
scala> words.groupBy(_.length).toSeq
res1: Seq[(Int, Seq[String])] = ArrayBuffer((2,List(is)), (4,List(food, good)), (3,List(the)))
```

It remains to swap the length and the list of words and to sort the result by increasing length. We can do this in any order: first sort, then swap; or first swap, then sort. The final code is

```
words
  .groupBy(_.length)
  .toSeq
  .sortBy { case (len, words) => len }
  .map { case (len, words) => (words, len) }
```

This code can be written somewhat shorter if we use the syntax _._1 for selecting the first part of the tuple, and .swap for swapping the two elements of the pair:

```
words.groupBy(_.length).toSeq.sortBy(_._1).map(_.swap)
```

However, the program may now be harder to read and to modify.

2.1.6 Reasoning about types of sequences

In Example 2.1.5.10 we have applied a chain of operations to a sequence. Let us add comments showing the type of the intermediate result after each operation:

```
words // Seq[String]
  .groupBy(_.length) // Map[Int, Seq[String]]
  .toSeq // Seq[ (Int, Seq[String]) ]
  .sortBy { case (len, words) => len } // Seq[ (Int, Seq[String]) ]
  .map { case (len, words) => (words, len) } // Seq[ (Seq[String], Int) ]
```

In computations like this, the Scala compiler verifies at each step that the operations are applied to values of the correct type.

For instance, `.sortBy` is defined for sequences but not for dictionaries, so it would be a type error to apply `.sortBy` to a dictionary without first converting it to a sequence using `.toSeq`. The type of the intermediate result after `.toSeq` is `Seq[ (Int, Seq[String]) ]`, and the `.sortBy` operation is applied to that sequence. So the sequence element matched by `{ case (len, words) => len }` is a tuple `(Int, Seq[String])`, which means that the pattern variables `len` and `words` must have types `Int` and `Seq[String]`

respectively. It would be a type error to use the sorting key function `{ case (len, words) => words }`: the sorting key can be an integer `len`, but not a string sequence `words` (because sorting by string sequences is not defined).

If we visualize how the type of the sequence should change at every step, we can more quickly understand how to implement the required task. Begin by writing down the intermediate types that would be needed during the computation:

```
words: Seq[String] // Need to group by word length.
Map[Int, Seq[String]]  // Need to sort by word length; can't sort a dictionary!
// Need to convert this dictionary to a sequence:
Seq[ (Int, Seq[String]) ] // Now sort this! Sorting does not change the type.
// It remains to swap the parts of all tuples in the sequence:
Seq[ (Seq[String], Int) ] // We are done.
```

Having written down these types, we are better assured that the computation can be done correctly. Writing the code becomes straightforward, since we are guided by the already known types of the intermediate results:

```
words.groupBy(_.length).toSeq.sortBy(_._1).map(_.swap)
```

This example illustrates the main benefits of reasoning about types: it gives direct guidance about how to organize the computation, together with a greater assurance in the correctness of the code.

2.1.7 Exercises: Tuples and collections

Exercise 2.1.7.1 Find all pairs i, j within $(0, 1, ..., 9)$ such that $i + 4 * j > i * j$.

Hint: use `.flatMap` and `.filter`.

Exercise 2.1.7.2 Same task as in Exercise 2.1.7.1, but for i, j, k and the condition $i + 4 * j + 9 * k > i * j * k$.

Exercise 2.1.7.3 Given two sequences `p: Seq[String]` and `q: Seq[Boolean]` of equal length, compute a `Seq[String]` with those elements of `p` for which the corresponding element of `q` is `true`.

Hint: use `.zip`, `.map`, `.filter`.

Exercise 2.1.7.4 Convert a `Seq[Int]` into a `Seq[(Int, Boolean)]` where the `Boolean` value is `true` when the element is followed by a larger value. For example, `Seq(1,3,2,4)` is to be converted into `Seq((1,true),(3,false),(2` (The last element, 4, has no following element.)

Exercise 2.1.7.5 Given `p: Seq[String]` and `q: Seq[Int]` of equal length, and assuming that elements of `q` do not repeat, compute a `Map[Int, String]` that maps numbers from `q` to their corresponding strings from `p`.

Exercise 2.1.7.6 Write the solution of Exercise 2.1.7.5 as a function with type parameters `P` and `Q` instead of the fixed types `String` and `Int`. Test it with `P = Boolean` and `Q = Set[Int]`.

Exercise 2.1.7.7 Given `p: Seq[String]` and `q: Seq[Int]` of equal length, compute a `Seq[String]` that contains the strings from `p` ordered according to the corresponding numbers from `q`. For example, if `p = Seq("a", "b", "c")` and `q = Seq(10, -1, 5)` then the result must be `Seq("b", "c", "a")`.

Exercise 2.1.7.8 Write the solution of Exercise 2.1.7.7 as a function with type parameter `S` instead of the fixed type `String`. The required type signature and a sample test:

```
def reorder[S](p: Seq[S], q: Seq[Int]): Seq[S] = ???

scala> reorder(Seq(6.0,2.0,8.0,4.0), Seq(20,10,40,30))
res0: Seq[Double] = List(2.0, 6.0, 4.0, 8.0)
```

Exercise 2.1.7.9 Given a `Seq[(String, Int)]` showing a list of purchased items (where item names may repeat), compute a `Map[String, Int]` showing the total counts: e.g. for the input

```
Seq(("apple", 2), ("pear", 3), ("apple", 5))
```

the output must be

```
Map("apple" -> 7, "pear" -> 3)
```

Implement this computation as a function with type parameter `S` instead of `String`.

Hint: use `.groupBy`, `.map`, `.sum`.

Exercise 2.1.7.10 Given a `Seq[List[Int]]`, compute a new `Seq[List[Int]]` where each inner list contains *three* largest elements from the initial inner list (or fewer than three if the initial inner list is shorter).

Hint: use `.map`, `.sortBy`, `.take`.

Exercise 2.1.7.11 **(a)** Given two sets `p:Set[Int]` and `q:Set[Int]`, compute a set of type `Set[(Int, Int)]` as the Cartesian product of the sets `p` and `q`; that is, the set of all pairs `(x, y)` where `x` is from `p` and `y` is from `q`.

(b) Implement this computation as a function with type parameters `I`, `J` instead of `Int`. The required type signature and a sample test:

```
def cartesian[I,J](p: Set[I], q: Set[J]): Set[(I, J)] = ???

scala> cartesian(Set("a", "b"), Set(10, 20))
res0: Set[(String, Int)] = Set((a,10), (a,20), (b,10), (b,20))
```

Hint: use `.flatMap` and `.map` on sets.

Exercise 2.1.7.12* Given a `Seq[Map[Person, Amount]]`, showing the amounts various people paid on each day, compute a `Map[Person, Seq[Amount]]`, showing the sequence of payments for each person. Assume that `Person` and `Amount` are type parameters. The required type signature and a sample test:

```
def payments[Person, Amount](data: Seq[Map[Person, Amount]]): Map[Person, Seq[Amount]] = ???

scala> payments(Seq(Map("Tarski" -> 10, "Church" -> 20), Map("Church" -> 100, "Gentzen" -> 40),
     Map("Tarski" -> 50)))
res0: Map[String, Seq[Int]] = Map(Genzten -> List(40), Church -> List(20, 100), Tarski -> List(10,
     50))
```

Hint: use `.flatMap`, `.groupBy`, `.mapValues` on dictionaries.

2.2 Converting a sequence into a single value

Until this point, we have been working with sequences using methods such as `.map` and `.zip`. These techniques are powerful but still insufficient for solving certain problems.

A simple computation that is impossible to do using `.map` is to compute the sum of a sequence of numbers. The standard library method `.sum` already does this; but we cannot implement `.sum` ourselves by using `.map`, `.zip`, or `.filter`. These operations always compute *new sequences*, while we need to compute a single value (the sum of all elements) from a sequence.

We have seen a few library methods such as `.count`, `.length`, and `.max` that compute a single value from a sequence; but we still cannot implement `.sum` using these methods. What we need is a more general way of converting a sequence to a single value, such that we could ourselves implement `.sum`, `.count`, `.max`, and other similar computations.

Another task not solvable with `.map`, `.sum`, etc., is to compute a floating-point number from a given sequence of decimal digits (including a "dot" character):

```
def digitsToDouble(ds: Seq[Char]): Double = ???

scala> digitsToDouble(Seq('2', '0', '4', '.', '5'))
res0: Double = 204.5
```

Why is it impossible to implement this function using `.map`, `.sum`, `.zip` and other methods we have seen so far? In fact, the same task for *integer* numbers (not for floating-point numbers) is solvable using `.length`, `.map`, `.sum`, and `.zip`:

```
def digitsToInt(ds: Seq[Int]): Int = {
  val n = ds.length
  // Compute a sequence of powers of 10, e.g. [1000, 100, 100, 1]
  val powers: Seq[Int] = (0 to n-1).map(k => math.pow(10, n-1-k).toInt)
  // Sum the powers of 10 with coefficients from `ds`.
  (ds zip powers).map { case (d, p) => d * p }.sum
}

scala> digitsToInt(Seq(2,4,0,5))
res0: Int = 2405
```

The computation can be written as the formula

$$r = \sum_{k=0}^{n-1} d_k * 10^{n-1-k} \quad .$$

The sequence of powers of 10 can be computed separately and "zipped" with the sequence of digits d_k. However, for floating-point numbers, the sequence of powers of 10 depends on the position of the "dot" character. Methods such as `.map` and `.zip` cannot compute a sequence whose next elements are not known in advance but depend on previous elements via a custom function.

2.2.1 Inductive definitions of aggregation functions

Mathematical induction is a general way of expressing the dependence of next values on previously computed values. To define a function from sequence to a single value (e.g. an aggregation function `f:Seq[Int] => Int`) by using mathematical induction, we need to specify two computations:

- (The **base case** of the induction.) We need to specify what value the function `f` returns for an empty sequence, `Seq()`. If the function is only defined for non-empty sequences, we need to specify what the function `f` returns for a one-element sequence such as `Seq(x)`, with any `x`.
- (The **inductive step**.) Assuming that the function `f` is already computed for some sequence `xs` (the **inductive assumption**), how to compute the function `f` for a sequence with one more element `x`? The sequence with one more element is written as `xs ++ Seq(x)`. So, we need to specify how to compute `f(xs ++ Seq(x))` assuming that `f(xs)` is already known.

Once these two computations are specified, the function `f` is defined (and can in principle be computed) for an arbitrary input sequence. This is how induction works in mathematics, and it works in the same way in functional programming. With this approach, the inductive definition of the method `.sum` looks like this:

- The sum of an empty sequence is 0. That is, `Seq().sum = 0`.
- If the result is already known for a sequence `xs`, and we have a sequence that has one more element `x`, the new result is equal to `xs.sum + x`. In code, this is `(xs ++ Seq(x)).sum = xs.sum + x`.

The inductive definition of the function `digitsToInt` is:

- For an empty sequence of digits, `Seq()`, the result is `0`. This is a convenient base case, even if we never call `digitsToInt` on an empty sequence.
- If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs ++ Seq(x)` with one more digit `x`, then

```
digitsToInt(xs ++ Seq(x)) = digitsToInt(xs) * 10 + x
```

Let us write inductive definitions for methods such as `.length`, `.max`, and `.count`:

- The length of a sequence:
  - for an empty sequence, `Seq().length = 0`
  - if `xs.length` is known then `(xs ++ Seq(x)).length = xs.length + 1`
- Maximum element of a sequence (undefined for empty sequences):
  - for a one-element sequence, `Seq(x).max = x`
  - if `xs.max` is known then `(xs ++ Seq(x)).max = math.max(xs.max, x)`
- Count the sequence elements satisfying a predicate `p`:
  - for an empty sequence, `Seq().count(p) = 0`
  - if `xs.count(p)` is known then `(xs ++ Seq(x)).count(p) = xs.count(p) + c`, where `c = 1` when `p(x) == true` and `c = 0` otherwise

There are two main ways of translating mathematical induction into code. The first way is to write a recursive function. The second way is to use a standard library function, such as `foldLeft` or `reduce`. Most often it is better to use the standard library functions, but sometimes the code is more transparent when using explicit recursion. So let us consider each of these ways in turn.

2.2.2 Implementing functions by recursion

A **recursive function** is any function that calls itself somewhere within its own body. The call to itself is the **recursive call**.

When the body of a recursive function is evaluated, it may repeatedly call itself with different arguments until the result value can be computed *without* any recursive calls. The last recursive call corresponds to the base case of the induction. It is an error if the base case is never reached, as in this example:

```
scala> def infiniteLoop(x: Int): Int = infiniteLoop(x+1)
infiniteLoop: (x: Int)Int

scala> infiniteLoop(2) // You will need to press Ctrl-C to stop this.
```

We translate mathematical induction into code by first writing a condition to decide whether we are in the base case or in the inductive step. As an example, consider how we would define `.sum` by recursion. The base case returns `0`, and the inductive step returns a value computed from the recursive call:

```
def sum(s: Seq[Int]): Int = if (s == Seq()) 0 else {
  val x = s.head // To split s = Seq(x) ++ xs, compute x
  val xs = s.tail //  and xs.
  sum(prev) + next // Call sum(...) recursively.
}
```

In this example, we use the `if/else` expression to separate the base case from the inductive step. In the inductive step, we split the given sequence `s` into a single-element sequence `Seq(x)`, the "head" of `s`, and the remainder ("tail") sequence `xs`. So, we split `s` as `s = Seq(x) ++ xs` rather than as `s = xs ++ Seq(x)`.

For computing the sum of a numerical sequence, the order of summation does not matter. However, the order of operations *will* matter for many other computational tasks. We need to choose whether the inductive step should split the sequence as `s = Seq(x) ++ xs` or as `s = xs ++ Seq(x)`, according to the task at hand.

Consider the implementation of `digitsToInt` according to the inductive definition shown in the previous subsection:

```
def digitsToInt(s: Seq[Int]): Int = if (s == Seq()) 0 else {
  val x = s.last  // To split s = xs ++ Seq(x), compute x
  val xs = s.take(s.length - 1) //  and xs.
  digitsToInt(xs) * 10 + x // Call digitstoInt(...) recursively.
}
```

In this example, it is important to split the sequence `s = xs ++ Seq(x)` in this order, and not in the order `Seq(x) ++ xs`. The reason is that digits increase their numerical value from right to left, so we need to multiply the value of the *left* subsequence, `digitsToInt(xs)`, by 10, in order to compute the correct result.

These examples show how mathematical induction is converted into recursive code. This approach often works but has two technical problems. The first problem is that the code will fail due to the "stack overflow" when the input sequence `s` is long enough. In the next subsection, we will see how this problem is solved (at least in some cases) using "tail recursion". The second problem is that each inductively defined function repeats the code for checking the base case and the code for splitting the sequence `s` into the subsequence `xs` and the extra element `x`. This repeated common code can be put into a library function, and the Scala library provides such functions. We will look at using them in Section 2.2.4.

2.2.3 Tail recursion

The code of `lengthS` will fail for large enough sequences. To see why, consider an inductive definition of the `.length` method as a function `lengthS`:

```
def lengthS(s: Seq[Int]): Int =
  if (s == Seq()) 0
  else 1 + lengthS(s.tail)

scala> lengthS((1 to 1000).toList)
res0: Int = 1000

scala> val s = (1 to 100000).toList
s: List[Int] = List(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, ...

scala> lengthS(s)
java.lang.StackOverflowError
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
  at .lengthS(<console>:12)
...
```

The problem is not due to insufficient main memory: we *are* able to compute and hold in memory the sequence `s`. The problem is with the code of the function `lengthS`. This function calls itself *inside* an expression `1 + lengthS(...)`. So we can visualize how the computer evaluates this code:

```
lengthS(Seq(1, 2, ..., 100000))
  = 1 + lengthS(Seq(2, ..., 100000))
  = 1 + (1 + lengthS(Seq(3, ..., 100000)))
  = ...
```

The function body of `lengthS` will evaluate the inductive step, that is, the "`else`" part of the "`if/else`", about 100000 times. Each time, the sub-expression with nested computations `1+(1+(...))` will get larger. This intermediate sub-expression needs to be held somewhere in memory, until at some point the function body goes into the base case and returns a value. When that happens, the entire intermediate sub-expression will contain about 100000 nested function calls still waiting to be evaluated. This sub-expression is held in a special area of memory called **stack memory**, where the not-yet-evaluated nested function calls are held in the order of their calls, as if on a "stack". Due to the way computer memory is managed, the stack memory has a fixed size and cannot grow automatically. So, when the intermediate expression becomes large enough, it causes an overflow of the stack memory, and the program may crash.

A way to solve this problem is to use a trick called **tail recursion**. Using tail recursion means rewriting the code so that all recursive calls occur at the end positions (at the "tails") of the function body. In other words, each recursive call must be *itself* the last computation in the function body. Recursive calls cannot be placed inside other computations.

As an example, we can rewrite the code of `lengthS` in this way:

```
def lengthT(s: Seq[Int], res: Int): Int =
  if (s == Seq()) res
  else lengthT(s.tail, 1 + res)
```

In this code, one of the branches of the `if/else` returns a fixed value without doing any recursive calls, while the other branch returns the result of recursive call to `lengthT(...)`. In the code of `lengthT`, recursive calls do not occur within sub-expressions such as `1 + lengthT(...)`, unlike in the code of `lengthS`.

It is not a problem that the recursive call to `lengthT` has some sub-expressions such as `1+res` as its arguments, because all these sub-expressions will be computed *before* `lengthT` is recursively called. The recursive call to `lengthT` is the *last* computation performed by this branch of the `if/else`. This shows that the code of `lengthT` is tail-recursive.

A tail-recursive function can have many `if/else` or `match/case` branches, with or without recursive calls; but all recursive calls must be always the last expressions returned.

The Scala compiler has a feature for checking automatically that a function's code is tail-recursive: the `@tailrec` annotation. If a function with a `@tailrec` annotation is not tail-recursive, or is not recursive at all, the program will not compile.

```
@tailrec def lengthT(s: Seq[Int], res: Int): Int =
  if (s == Seq()) res
  else lengthT(s.tail, 1 + res)
```

Let us trace the evaluation of this function on a short example:

```
lengthT(Seq(1,2,3), 0)
  = lengthT(Seq(2, 3), 1 + 0) // = lengthT(Seq(2, 3), 1)
  = lengthT(Seq(3), 1 + 1) // = lengthT(Seq(3), 2)
  = lengthT(Seq(), 1 + 2) // = lengthT(Seq(), 3)
  = 3
```

The sub-expressions such as `1 + 2` are computed each time *before* each recursive call to `lengthT`. Because of that, sub-expressions do not grow within the stack memory. This is the main benefit of tail recursion.

How did we rewrite the code of `lengthS` to obtain the tail-recursive code of `lengthT`? An important

difference between `lengthS` and `lengthT` is the additional argument, `res`, called the **accumulator argument**. This argument is equal to an intermediate result of the computation. The next intermediate result (`1 + res`) is computed and passed on to the next recursive call via the accumulator argument. In the base case of the recursion, the function now returns the accumulated result, `res`, rather than `0`, because at that time the computation is finished.

Rewriting code by adding an accumulator argument to achieve tail recursion is called the **accumulator technique** or the "accumulator trick".

One consequence of using the accumulator trick is that the function `lengthT` now always needs a value for the accumulator argument. However, our goal is to implement a function such as `length(s)` with just one argument, `s:Seq[Int]`. We can define `length(s) = lengthT(s, ???)` if we supply an initial accumulator value. The correct initial value for the accumulator is `0`, since in the base case (an empty sequence `s`) we need to return `0`.

So, a tail-recursive implementation of `lengthT` requires us to define *two* functions: the tail-recursive `lengthT` and an "adapter" function that will set the initial value of the accumulator argument. To emphasize that `lengthT` is a helper function, one could define it *inside* the adapter function:

```
def length[A](s: Seq[A]): Int = {
  @tailrec def lengthT(s: Seq[A], res: Int): Int = {
    if (s == Seq()) res
    else lengthT(s.tail, 1 + res)
  }
  lengthT(s, 0)
}
```

With this code, users will not be able to call `lengthT` directly, since it is only visible within the body of the `length` function.

Another possibility in Scala is to use a **default value** for the `res` argument:

```
@tailrec def length[A](s: Seq[A], res: Int = 0): Int =
  if (s == Seq()) res
  else length(s.tail, 1 + res)
```

In Scala, giving a default value for a function argument is the same as defining *two* functions: one with that argument and one without. For example, the syntax

```
def f(x: Int, y: Boolean = false): Int = ... // Function body.
```

is equivalent to defining two functions (with the same name),

```
def f(x: Int, y: Boolean) = ... // Function body.
def f(x: Int): Int = f(Int, false)
```

Using a default argument value, we can define the tail-recursive helper function and the adapter function at once, making the code shorter.

The accumulator trick works in a large number of cases, but it may be far from obvious how to introduce the accumulator argument, what its initial value must be, and how to define the induction step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the following mathematical property of the expression being computed:

$$1 + (1 + (1 + (... + 1))) = (((1 + 1) + 1) + ...) + 1 \quad .$$

This property is called the **associativity law** of addition. Due to this law, the computation can be rearranged so that additions associate to the left. In code, it means that intermediate expressions are computed immediately before making the recursive calls; this avoids the growth of the intermediate expressions.

Usually, the accumulator trick works because some associativity law is present. In that case, we are able to rearrange the order of recursive calls so that these calls always occur outside all other sub-

expressions, – that is, in tail positions. However, not all computations obey a suitable associativity law. Even if a code rearrangement exists, it may not be immediately obvious how to find it.

As an example, consider a tail-recursive re-implementation of the function `digitsToInt` from the previous subsection, where the recursive call is within a sub-expression `digitsToInt(xs) * 10 + x`. To transform the code into a tail-recursive form, we need to rearrange the main computation,

$$r = d_{n-1} + 10 * (d_{n-2} + 10 * (d_{n-3} + 10 * (...d_0))) \quad ,$$

so that the operations group to the left. We can do this by rewriting r as

$$r = ((d_0 * 10 + d_1) * 10 + ...) * 10 + d_{n-1} \quad .$$

It follows that the digit sequence `s` must be split into the *leftmost* digit and the rest, `s = s.head ++ s.tail`. So, a tail-recursive implementation of the above formula is

```
@tailrec def fromDigits(s: Seq[Int], res: Int = 0): Int =
  // ‘res‘ is the accumulator.
  if (s == Seq()) res
  else fromDigits(s.tail, 10 * res + s.head)
```

Despite a certain similarity between this code and the code of `digitsToInt` from the previous subsection, the implementation `fromDigits` cannot be directly derived from the inductive definition of `digitsToInt`. One needs a separate proof that `fromDigits(s, 0)` computes the same result as `digitsToInt(s)`. The proof follows from the following property.

Statement 2.2.3.1 For any `xs: Seq[Int]` and `r: Int`, we have

```
fromDigits(xs, r) = digitsToInt(xs) + r * math.pow(10, s.length)
```

Proof We prove this by induction. Let us use a short notation for sequences, $[1,2,3]$ instead of `Seq(1, 2, 3)`, and temporarily write $d(s)$ instead of `digitsToInt(s)` and $f(s,r)$ instead of `fromDigitsT(s, r)`. Then an inductive definition of $f(s,r)$ is

$$f([],r) = r \quad , \qquad f([x]{+}{+}s,r) = f(s, 10 * r + x) \quad . \tag{2.1}$$

Denoting by $|s|$ the length of a sequence s, we reformulate Statement 2.2.3.1 as

$$f(s,r) = d(s) + r * 10^{|s|} \quad , \tag{2.2}$$

We prove Eq. (2.2) by induction. To prove the base case $s = []$, we have $f([],r) = r$ and $d([]) + r * 10^0 = r$ since $d([]) = 0$ and $|s| = 0$. The resulting equality $r = r$ proves the base case.

To prove the inductive step, we assume that Eq. (2.2) holds for a given sequence s; then we need to prove that

$$f([x]{+}{+}s,r) = d([x]{+}{+}s) + r * 10^{|s|+1} \quad . \tag{2.3}$$

To prove this, we transform the left-hand side and the right-hand side separately, hoping that we will obtain the same expression. The left-hand side of Eq. (2.3):

$$\begin{aligned} & f([x]{+}{+}s,r) \\ \text{use Eq. (2.1)} : \quad & = f(s, 10 * r + x) \\ \text{use Eq. (2.2)} : \quad & = d(s) + (10 * r + x) * 10^{|s|} \quad . \end{aligned}$$

The right-hand side of Eq. (2.3) contains $d([x]{+}{+}s)$, which we somehow need to simplify. Assuming that $d(s)$ correctly calculates a number from its digits, we can use the basic property of decimal notation, which is that a digit x in front of n other digits has the value $x * 10^n$. This property can be formulated as

$$d([x]{+}{+}s) = x * 10^{|s|} + d(s) \quad . \tag{2.4}$$

So, the right-hand side of Eq. (2.3) can be rewritten as

$$\begin{aligned} & d([x]{+}{+}s) + r * 10^{|s|+1} \\ \text{use Eq. (2.4)} : \quad & = x * 10^{|s|} + d(s) + r * 10^{|s|+1} \\ \text{factor out } 10^{|s|} : \quad & = d(s) + (10 * r + x) * 10^{|s|} \quad . \end{aligned}$$

Now we have transformed both sides of Eq. (2.3) to the same expression.

We have not yet proved that the function d satisfies the property in Eq. (2.4). The proof uses induction and begins by writing the code of d in a short notation,

$$d([]) = 0 \quad , \qquad d(s{+}{+}[y]) = d(s) * 10 + y \quad . \tag{2.5}$$

The base case is Eq. (2.4) with $s = []$. It is proved by

$$x = d([]{+}{+}[x]) = d([x]{+}{+}[]) = x * 10^0 + d([]) = x \quad .$$

The induction step assumes Eq. (2.4) for a given x and a given sequence s, and needs to prove that for any y, the same property holds with $s{+}{+}[y]$ instead of s:

$$d([x]{+}{+}s{+}{+}[y]) = x * 10^{|s|+1} + d(s{+}{+}[y]) \quad . \tag{2.6}$$

The left-hand side of Eq. (2.6) is transformed into its right-hand side like this:

$$\begin{aligned} & d([x]{+}{+}s{+}{+}[y]) \\ \text{use Eq. (2.5)} : \quad & = d([x]{+}{+}s) * 10 + y \\ \text{use Eq. (2.4)} : \quad & = \left(x * 10^{|s|} + d(s) \right) * 10 + y \\ \text{expand parentheses} : \quad & = x * 10^{|s|+1} + d(s) * 10 + y \\ \text{use Eq. (2.5)} : \quad & = x * 10^{|s|+1} + d(s{+}{+}[y]) \quad . \end{aligned}$$

This demonstrates Eq. (2.6) and so concludes the proof.

2.2.4 Implementing generic aggregation (`foldLeft`)

An **aggregation** converts a sequence of values into a single value. In general, the type of the result value may be different from the type of elements in the sequence. To describe this general situation, we introduce type parameters, `A` and `B`, so that the input sequence is of type `Seq[A]` and the aggregated value is of type `B`. Then an inductive definition of any aggregation function `f: Seq[A] => B` looks like this:

- (Base case.) For an empty sequence, `f(Seq()) = b0` where `b0:B` is a given value.
- (Induction step.) Assuming that `f(xs) = b` is already computed, we define `f(xs ++ Seq(x)) = g(x, b)` where `g` is a given function with type signature `g:(A, B) => B`.

Then the code implementing `f` is written using recursion:

```
def f[A, B](s: Seq[A]): B =
  if (s == Seq()) b0
  else g(s.last, f(s.take(s.length - 1)))
```

We can now refactor this code into a generic utility function, by making `b0` and `g` into parameters. A possible implementation is

```
def f[A, B](s: Seq[A], b: B, g: (A, B) => B): B =
  if (s == Seq()) b
  else g(s.last, f(s.take(s.length - 1), b, g)
```

However, this implementation is not tail-recursive. Applying `f` to a sequence of, say, three elements, `Seq(x, y, z)`, will create an intermediate expression `g(z, g(y, g(x, b)))`. This expression will grow with the length of `s`, which is not acceptable. To rearrange the computation into a tail-recursive form, we need to start the base case at the innermost call `g(x, b)`, then compute `g(y, g(x, b))` and continue. In other words, we need to traverse the sequence starting from its *leftmost* element `x`, rather than starting from the right. So, instead of splitting the sequence `s` into `s.last ++ s.take(s.length - 1)` as we did in the code of `f`, we need to split `s` into `s.head ++ s.tail`. Let us also exchange the order of the arguments of `g`, in order to be more consistent with the way this code is implemented in the Scala library. The resulting code is now tail-recursive:

```
@tailrec def leftFold[A, B](s: Seq[A], b: B, g: (B, A) => B): B =
  if (s == Seq()) b
  else leftFold(s.tail, g(b, s.head), g)
```

This function is called a "left fold" because it aggregates (or "folds") the sequence starting from the leftmost element.

In this way, we have defined a general method of computing any inductively defined aggregation function on a sequence. The function `leftFold` implements the logic of aggregation defined via mathematical induction. Using `leftFold`, we can write concise implementations of methods such as `.sum` or `.max`, and of many other similar aggregation methods. The method `leftFold` already contains all the code necessary to set up the base case and the induction step. The programmer just needs to specify the expressions for the initial value `b` and for the updater function `g`.

As a first example, let us use `leftFold` for implementing the `.sum` method:

```
def sum(s: Seq[Int]): Int = leftFold(s, 0, { (x, y) => x + y })
```

To understand in detail how `leftFold` works, let us trace the evaluation of this function when applied to `Seq(1, 2, 3)`:

```
sum(Seq(1, 2, 3)) = leftFold(Seq(1, 2, 3), 0, g)
  // Here, g = { (x, y) => x + y }, so g(x, y) = x + y
  = leftFold(Seq(2, 3), g(0, 1), g) // g (0, 1) = 1
  = leftFold(Seq(2, 3), 1, g) // now expand the code of leftFold
  = leftFold(Seq(3), g(1, 2), g) // g(1, 2) = 3; expand the code
  = leftFold(Seq(), g(3, 3), g) // g(3, 3) = 6; expand the code
  = 6
```

The second argument of `leftFold` is the accumulator argument. The initial value of the accumulator is specified when first calling `leftFold`. At each iteration, the new accumulator value is computed by calling the updater function `g`, which uses the previous accumulator value and the value of the next sequence element.

To visualize the process of evaluation, it is convenient to write a table showing the sequence elements and the accumulator values as they are updated:

Current element `x`	Old accumulator value	New accumulator value
1	0	1
2	1	3
3	3	6

In general, the type of the accumulator value can be different from the type of the sequence ele-

ments. An example is an implementation of `count`:

```
def count[A](s: Seq[A], p: A => Boolean): Int =
  leftFold(s, 0, { (x, y) => x + (if (p(y)) 1 else 0) })
```

The accumulator is of type `Int`, while the sequence elements can have an arbitrary type, parameterized by `A`. The aggregation function `leftFold` works in the same way for all types of accumulators and all types of sequence elements.

In Scala's standard library, the `.foldLeft` method has a different type signature and, in particular, requires its arguments to be in separate argument groups. For comparison, the implementation of `sum` using our `leftFold` function is

```
def sum(s: Seq[Int]): Int = leftFold(s, 0, { (x, y) => x + y })
```

With the Scala library's `.foldLeft` method, the code is written as

```
def sum(s: Seq[Int]): Int = s.foldLeft(0){ (x, y) => x + y }
```

This syntax makes it more convenient to write nameless functions as arguments, since the updater argument of `.foldLeft` is separated from other arguments by curly braces. We will use the standard `.foldLeft` method from now on; the `leftFold` function was implemented only as an illustration.

The method `.foldLeft` is available in the Scala standard library for all collections, including dictionaries and sets. It is safe to use `.foldLeft`, in the sense that no stack overflows will occur even for very large sequences.

The Scala library contains several other methods similar to `.foldLeft`, such as `.foldRight` and `.reduce`. (However, `.foldRight` is not tail-recursive!)

2.2.5 Solved examples: using `foldLeft`

It is important to gain experience using the `.foldLeft` method.

Example 2.2.5.1 Use `.foldLeft` for implementing the `max` function for integer sequences. Return the special value `Int.MinValue` for empty sequences.

Solution: Write an inductive formulation of the `max` function:

- (Base case.) For an empty sequence, return `Int.MinValue`.
- (Inductive step.) If `max` is already computed on a sequence `xs`, say `max(xs) = b`, the value of `max` on a sequence `xs ++ Seq(x)` is `math.max(b,x)`.

Now we can write the code:

```
def max(s: Seq[Int]): Int =
  s.foldLeft(Int.MinValue){ (b, x) => math.max(b, x) }
```

If we are sure that the function will never be called on empty sequences, we can implement `max` in a simpler way by using the `.reduce` method:

```
def max(s: Seq[Int]): Int = s.reduce { (x, y) => math.max(x, y) }
```

Example 2.2.5.2 Implement the `count` method on sequences of type `Seq[A]`.

Solution: Using the inductive definition of the function `count` as shown in Section 2.2.1, we can write the code as

```
def count[A](s: Seq[A], p: A => Boolean): Int =
  s.foldLeft(0){ (b, x) => b + (if (p(x)) 1 else 0) }
```

Example 2.2.5.3 Implement the function `digitsToInt` using `.foldLeft`.

Solution: The inductive definition of `digitsToInt` is directly translated into code:

```
def digitsToInt(d: Seq[Int]): Int = d.foldLeft(0){ (n, x) => n * 10 + x }
```

Example 2.2.5.4 For a given non-empty sequence `xs: Seq[Double]`, compute the minimum, the maximum, and the mean as a tuple $(x_{\min}, x_{\max}, x_{\text{mean}})$. The sequence should be traversed only once, i.e. we may call `xs.foldLeft` only once.

Solution: Without the requirement of using a single traversal, we would write

```
(xs.min, xs.max, xs.sum / xs.length)
```

However, this code traverses `xs` at least three times, since each of the aggregations `xs.min`, `xs.max`, and `xs.sum` iterates over `xs`. We need to combine the four inductive definitions of `min`, `max`, `sum`, and `length` into a single inductive definition of some function. What is the type of that function's return value? We need to accumulate intermediate values of *all four* numbers (`min`, `max`, `sum`, and `length`) in a tuple. So the required type of the accumulator is `(Double, Double, Double, Double)`. To avoid repeating a very long type expression, we can define a type alias for it, say, `D4`:

```
scala> type D4 = (Double, Double, Double, Double)
defined type alias D4
```

The updater function must update each of the four numbers according to the definitions of their inductive steps:

```
def update(p: D4, x: Double): D4 = p match {
  case (min, max, sum, length) =>
   (math.min(x, min), math.max(x, max), x + sum, length + 1)
}
```

Now we can write the code of the required function:

```
def f(xs: Seq[Double]): (Double, Double, Double) = {
  val init: D4 = (Double.PositiveInfinity, Double.NegativeInfinity, 0, 0)
  val (min, max, sum, length) = xs.foldLeft(init)(update)
  (min, max, sum/length)
}

scala> f(Seq(1.0, 1.5, 2.0, 2.5, 3.0))
res0: (Double, Double, Double) = (1.0,3.0,2.0)
```

Example 2.2.5.5* Implement the function `digitsToDouble` using `.foldLeft`. The argument is of type `Seq[Char]`. For example,

```
digitsToDouble(Seq('3', '4', '.', '2', '5')) = 34.25
```

Assume that all characters are either digits or the dot character (so, negative numbers are not supported).

Solution: The evaluation of a `.foldLeft` on a sequence of digits will visit the sequence from left to right. The updating function should work the same as in `digitsToInt` until a dot character is found. After that, we need to change the updating function. So, we need to remember whether a dot character has been seen. The only way for `.foldLeft` to "remember" any data is to hold that data in the accumulator value. We can choose the type of the accumulator according to our needs. So, for this task we can choose the accumulator to be a *tuple* that contains, for instance, the floating-point result constructed so far and a `Boolean` flag showing whether we have already seen the dot character.

To visualize what `digitsToDouble` must do, let us consider how the evaluation of `digitsToDouble(Seq('3', '4', '.', '2', '5'))` should go. We can write a table showing the intermediate result at each iteration.

This will hopefully help us figure out what the accumulator and the updater function $g(...)$ must be:

Current digit c	**Previous result** n	**New result** $n' = g(n, c)$
`'3'`	0.0	3.0
`'4'`	3.0	34.0
`'.'`	34.0	34.0
`'2'`	34.0	34.2
`'5'`	34.2	34.25

Until the dot character is found, the updater function multiplies the previous result by 10 and adds the current digit. After the dot character, the updater function must add to the previous result the current digit divided by a factor that represents increasing powers of 10. In other words, the update computation $n' = g(n, c)$ must be defined by these formulas:

- Before the dot character: $g(n, c) = n * 10 + c$.
- After the dot character: $g(n, c) = n + \frac{c}{f}$, where f is 10, 100, 1000, etc., for each subsequent digit.

The updater function g has only two arguments: the current digit and the previous accumulator value. So, the changing factor f must be *part of* the accumulator value, and must be multiplied by 10 at each digit after the dot. If the factor f is not a part of the accumulator value, the function g will not have enough information for computing the next accumulator value correctly. So, the updater computation must be $n' = g(n, c, f)$, not $n' = g(n, c)$.

For this reason, we choose the accumulator type as a tuple `(Double, Boolean, Double)` where the first number is the result n computed so far, the `Boolean` flag indicates whether the dot was already seen, and the third number is f, that is, the power of 10 by which the current digit will be divided if the dot was already seen. Initially, the accumulator tuple will be equal to `(0.0, false, 10.0)`. Then the updater function is implemented like this:

```
def update(acc: (Double, Boolean, Double), c: Char): (Double, Boolean, Double) =
acc match { case (n, flag, factor) =>
  if (c == '.') (n, true, factor) // Set flag to ‘true‘ if dot character.
  else {
    val digit = c - '0'
    if (flag) // This digit is after the dot. Update ‘factor‘.
      (n + digit/factor, flag, factor * 10)
    else // This digit is before the dot.
      (n * 10 + digit, flag, factor)
  }
}
```

Now we can implement `digitsToDouble` as follows,

```
def digitsToDouble(d: Seq[Char]): Double = {
  val initAccumulator = (0.0, false, 10.0)
  val (n, _, _) = d.foldLeft(initAccumulator)(update)
  n
}

scala> digitsToDouble(Seq('3', '4', '.', '2', '5'))
res0: Double = 34.25
```

The result of calling `d.foldLeft` is a tuple `(n, flag, factor)`, in which only the first part, `n`, is needed. In Scala's pattern matching expressions, the underscore symbol is used to denote the pattern variables whose values are not needed in the subsequent code. We could extract the first part using the accessor method `._1`, but the code is more readable if we show all parts of the tuple as `(n, _, _)`.

Example 2.2.5.6 Implement the `.map` method for sequences by using `.foldLeft`. The input sequence should be of type `Seq[A]`, and the output sequence of type `Seq[B]`, where `A` and `B` are type parameters. Here are the required type signature of the function and a sample test:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] = ???

scala> map(List(1, 2, 3)){ x => x * 10 }
res0: Seq[Int] = List(10, 20, 30)
```

Solution: The required code should build a new sequence by applying the function `f` to each element. How can we build a new sequence using `.foldLeft`? The evaluation of `.foldLeft` consists of iterating over the input sequence and accumulating some result value, which is updated at each iteration. Since the result of a `.foldLeft` is always equal to the last computed accumulator value, it follows that the new sequence should *be* the accumulator value. So, we need to update the accumulator by appending the value `f(x)`, where `x` is the current element of the input sequence. We can append elements to sequences using the `:+` operation:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] =
  xs.foldLeft(Seq[B]()){ (acc, x) => acc :+ f(x) }
```

The operation `acc :+ f(x)` is equivalent to `acc ++ Seq(f(x))` but is shorter to write.

Example 2.2.5.7 Implement a function `toPairs` that converts a sequence of type `Seq[A]` to a sequence of pairs, `Seq[(A, A)]`, by putting together each pair of adjacent elements. If the initial sequence has an odd number of elements, a given default value of type `A` is used:

```
def toPairs[A](xs: Seq[A], default: A): Seq[(A, A)] = ???

scala> toPairs(Seq(1, 2, 3, 4, 5, 6), -1)
res0: Seq[(Int, Int)] = List((1,2), (3,4), (5,6))

scala> toPairs(Seq("a", "b", "c"), "<nothing>")
res1: Seq[(String, String)] = List((a,b), (c,<nothing>))
```

Solution: We need to use `.foldLeft` to accumulate a sequence of pairs. However, we iterate over elements of the input sequence one by one. So, a new pair can be added only once every two iterations. The accumulator needs to hold the information about the current iteration being even or odd. For odd-numbered iterations, the accumulator also needs to store the previous element that is still waiting for its pair. Therefore, we choose the type of the accumulator to be a tuple `(Seq[(A, A)], Seq(A))`. The first sequence is the intermediate result, and the second sequence is the "remainder": it holds the previous element for odd-numbered iterations and is empty for even-numbered iterations. Initially, the accumulator should be empty. A trace of the accumulator updates is shown in this table:

Current element `x`	Previous accumulator	Next accumulator
`"a"`	`(Seq(), Seq())`	`(Seq(), Seq("a"))`
`"b"`	`(Seq(), Seq("a"))`	`(Seq(("a","b")), Seq())`
`"c"`	`(Seq(("a","b")), Seq())`	`(Seq(("a","b")), Seq("c"))`

Now it becomes clear how to implement the updater function. The code calls `.foldLeft` and then performs some post-processing to make sure we create the last pair in case the last iteration is odd-numbered, i.e. when the "remainder" is not empty after `.foldLeft` is finished. In this implementation, we use pattern matching to decide whether a sequence is empty:

```
def toPairs[A](xs: Seq[A], default: A): Seq[(A, A)] = {
  type Acc = (Seq[(A, A)], Seq[A]) // Type alias, for brevity.
  def init: Acc = (Seq(), Seq())
```

```
  def updater(acc: Acc, x: A): Acc = acc match {
    case (result, Seq()) => (result, Seq(x))
    case (result, Seq(prev)) => (result ++ Seq((prev, x)), Seq())
  }
  val (result, remainder) = xs.foldLeft(init)(updater)
  // May need to append the last element to the result.
  remainder match {
    case Seq() => result
    case Seq(x) => result ++ Seq((x, default))
  }
}
```

This code shows examples of partial functions that are applied safely. One of these partial functions is used in the expression

```
remainder match {
  case Seq() => ...
  case Seq(a) => ...
}
```

This code works when `remainder` is empty or has length 1, but fails for longer sequences. So it is safe to apply the partial function as long as it is used on sequences of length at most 1, which is indeed the case for the code of `toPairs`.

2.2.6 Exercises: Using `foldLeft`

Exercise 2.2.6.1 Implement a function `fromPairs` that performs the inverse transformation to the `toPairs` function defined in Example 2.2.5.7. The required type signature and a sample test:

```
def fromPairs[A](xs: Seq[(A, A)]): Seq[A] = ???

scala> fromPairs(Seq((1, 2), (3, 4)))
res0: Seq[Int] = List(1, 2, 3, 4)
```

Hint: This can be done with `.foldLeft` or with `.flatMap`.

Exercise 2.2.6.2 Implement the `flatten` method for sequences by using `.foldLeft`. The required type signature and a sample test:

```
def flatten[A](xxs: Seq[Seq[A]]): Seq[A] = ???

scala> flatten(Seq(Seq(1, 2, 3), Seq(), Seq(4)))
res0: Seq[Int] = List(1, 2, 3, 4)
```

Exercise 2.2.6.3 Use `.foldLeft` to implement the `zipWithIndex` method for sequences. The required type signature and a sample test:

```
def zipWithIndex[A](xs: Seq[A]): Seq[(A, Int)] = ???

scala> zipWithIndex(Seq("a", "b", "c", "d"))
res0: Seq[String] = List((a, 0), (b, 1), (c, 2), (d, 3))
```

Exercise 2.2.6.4 Use `.foldLeft` to implement a function `filterMap` that combines `.map` and `.filter` for sequences. The required type signature and a sample test:

```
def filterMap[A, B](xs: Seq[A])(pred: A => Boolean)(f: A => B): Seq[B] = ???

scala> filterMap(Seq(1, 2, 3, 4)) { x => x > 2 } { x => x * 10 }
res0: Seq[Int] = List(30, 40)
```

Exercise 2.2.6.5* Split a sequence into subsequences ("batches") of length not larger than a given maximum length n. The required type signature and a sample test are:

```
def batching[A](xs: Seq[A], size: Int): Seq[Seq[A]] = ???

scala> batching(Seq("a", "b", "c", "d"), 2)
res0: Seq[Seq[String]] = List(List(a, b), List(c, d))

scala> batching(Seq(1, 2, 3, 4, 5, 6, 7), 3)
res1: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7))
```

Exercise 2.2.6.6* Split a sequence into batches by "weight" computed via a given function. The total weight of items in any batch should not be larger than a given maximum weight. The required type signature and a sample test:

```
def byWeight[A](xs: Seq[A], maxW: Double)(w: A => Double): Seq[Seq[A]] = ???

scala> byWeight((1 to 10).toList, 5.75){ x => math.sqrt(x) }
res0: Seq[Seq[Int]] = List(List(1, 2, 3), List(4, 5), List(6, 7), List(8), List(9), List(10))
```

Exercise 2.2.6.7* Use `.foldLeft` to implement a `groupBy` function. The required type signature and a sample test:

```
def groupBy[A, K](xs: Seq[A])(by: A => K): Map[K, Seq[A]] = ???

scala> groupBy(Seq(1, 2, 3, 4, 5)){ x => x % 2 }
res0: Map[Int, Seq[Int]] = Map(1 -> List(1, 3, 5), 0 -> List(2, 4))
```

Hints:

The accumulator should be of type `Map[K, Seq[A]]`.

To work with dictionaries, you will need to use the methods `.getOrElse` and `.updated`. The method `.getOrElse` fetches a value from a dictionary by key, and returns the given default value if the dictionary does not contain that key:

```
scala> Map("a" -> 1, "b" -> 2).getOrElse("a", 300)
res0: Int = 1

scala> Map("a" -> 1, "b" -> 2).getOrElse("c", 300)
res1: Int = 300
```

The method `.updated` produces a new dictionary that contains a new value for the given key, whether or not that key already exists in the dictionary:

```
scala> Map("a" -> 1, "b" -> 2).updated("c", 300) // Key is new.
res0: Map[String,Int] = Map(a -> 1, b -> 2, c -> 300)

scala> Map("a" -> 1, "b" -> 2).updated("a", 400) // Key already exists.
res1: Map[String,Int] = Map(a -> 400, b -> 2)
```

2.3 Converting a single value into a sequence

An aggregation converts or "folds" a sequence into a single value; the opposite operation ("unfolding") converts a single value into a sequence. An example of this task is to compute the sequence of decimal digits for a given integer:

```
def digitsOf(x: Int): Seq[Int] = ???
```

```
scala> digitsOf(2405)
res0: Seq[Int] = List(2, 4, 0, 5)
```

We cannot implement this function using `.map`, `.zip`, or `.foldLeft`, because these methods may be applied only if we *already have* a sequence. A new sequence can be created, e.g., via the expression `(1 to n)`, but we do not know in advance how long the required sequence must be. The length of the required sequence is determined by a condition that we cannot easily evaluate in advance.

A general "unfolding" operation requires us to build a sequence whose length is not determined in advance. This kind of sequence is called an **stream**. A stream is a sequence whose elements are computed only when necessary (unlike sequences such as a `List` or an `Array`, whose elements are all computed in advance and stored). The unfolding operation will keep computing the next element; this creates a stream. We can then apply `.takeWhile` to the stream, in order to stop it when a certain condition holds. Finally, if required, the truncated stream may be converted to a list or to another type of sequence. In this way, we can generate a sequence of initially unknown length according to the given requirements.

A general stream-producing function `Stream.iterate` is available in the Scala library. This function has two arguments, the initial value and a function that computes the next value from the previous one:

```
scala> Stream.iterate(2){ x => x + 10 }
res0: Stream[Int] = Stream(2, ?)
```

The stream is ready to start computing the next elements of the sequence (so far, only the first element, 2, has been computed). In order to see the next elements, we need to stop the stream at a finite size and then convert the result to a list:

```
scala> Stream.iterate(2){ x => x + 10 }.take(6).toList
res1: List[Int] = List(2, 12, 22, 32, 42, 52)
```

If we try to evaluate `.toList` on a stream without first limiting its size via `.take` or `.takeWhile`, the program will keep producing more and more elements of the stream, until it runs out of memory and crashes.

Streams are similar to sequences, and methods such as `.map`, `.filter`, `.flatMap` are also defined for streams. For instance, we can use the method `.drop` that skips a given number of initial elements:

```
scala> Seq(10, 20, 30, 40, 50).drop(3)
res2: Seq[Int] = List(40, 50)

scala> Stream.iterate(2){ x => x + 10 }.drop(3)
res3: Stream[Int] = Stream(32, ?)
```

This example shows that in order to evaluate `.drop(3)`, the stream had to compute its elements up to 32 (but the subsequent elements are still not computed).

To figure out the code for `digitsOf`, we first write this function as a mathematical formula. To compute the sequence of digits for, say, $n = 2405$, we need to divide n repeatedly by 10, getting a sequence n_k of intermediate numbers ($n_0 = 2405$, $n_1 = 240$, ...) and the corresponding sequence of last digits, $n_k \bmod 10$ (in this example: 5, 0, ...). The sequence n_k is defined using mathematical induction:

- Base case: $n_0 = n$, where n is the given initial integer.
- Inductive step: $n_{k+1} = \left\lfloor \frac{n_k}{10} \right\rfloor$ for $k = 1, 2, ...$

Here $\left\lfloor \frac{n_k}{10} \right\rfloor$ is the mathematical notation for the integer division by 10.

Let us trace the evaluation of the sequence n_k for $n = 2405$:

$k =$	0	1	2	3	4	5	6
$n_k =$	2405	240	24	2	0	0	0
$n_k \bmod 10 =$	5	0	4	2	0	0	0

The numbers n_k will remain all zeros after $k = 4$. It is clear that the useful part of the sequence is before it becomes all zeros. In this example, the sequence n_k needs to be stopped at $k = 4$. The sequence of digits then becomes $[5, 0, 4, 2]$, and we need to reverse it to obtain $[2, 4, 0, 5]$. For reversing a sequence, the Scala library has the standard method `.reverse`. So the code is

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n) { nk => nk / 10 }
      .takeWhile { nk => nk != 0 }
      .map { nk => nk % 10 }
      .toList.reverse
  }
```

By writing nameless functions such as `{ nk => nk % 10 }` in a shorter syntax such as `(_ % 10)`, we can shorten the code of `digitsOf`:

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n)(_ / 10)
      .takeWhile(_ != 0)
      .map(_ % 10)
      .toList.reverse
  }
```

The type signature of the method `Stream.iterate` can be written as

```
def iterate[A](init: A)(next: A => A): Stream[A]
```

This shows how `Stream.iterate` constructs a sequence defined by mathematical induction. The base case is the first value, `init`, and the inductive step is a function, `next`, that computes the next element from the previous one. It is a flexible way of generating sequences whose length is not determined in advance.

2.4 Transforming a sequence into another sequence

We have seen methods such as `.map` and `.zip` that transform sequences into sequences. However, these methods cannot express a general transformation where the elements of the new sequence are defined by induction, depending on previous elements. An example of a task of this kind is to compute the partial sums of a given sequence x_i:

$$b_k = \sum_{i=0}^{k-1} x_i \quad .$$

This formula defines $b_0 = 0$, $b_1 = x_0$, $b_2 = x_0 + x_1$, $b_3 = x_0 + x_1 + x_2$, etc. A definition via mathematical induction may be written like this,

- (Base case.) $b_0 = 0$.
- (Induction step.) Given b_k, we define $b_{k+1} = b_k + x_k$ for $k = 0, 1, 2, ...$

The Scala library method `.scanLeft` implements a general sequence-to-sequence transformation defined in this way. The code implementing the partial sums is

```
def partialSums(xs: Seq[Int]): Seq[Int] = xs.scanLeft(0){ (x, y) => x + y }

scala> partialSums(Seq(1, 2, 3, 4))
res0: Seq[Int] = List(0, 1, 3, 6, 10)
```

The first argument of `.scanLeft` is the base case, and the second argument is an updater function describing the induction step. In general, the type of elements of the second sequence is different from that of the first sequence. The updater function takes an element of the first sequence and a previous element of the second sequence, and returns the next element of the second sequence. Note that the result of `.scanLeft` is one element longer than the original sequence, because the base case provides an initial value.

Until now, we have seen that `.foldLeft` is sufficient to re-implement almost every method that work on sequences, such as `.map`, `.filter`, or `.flatten`. The method `.scanLeft` can be also implemented via `.foldLeft`. In the implementation, the accumulator contains the previous element of the second sequence together with a growing fragment of that sequence, which is updated as we iterate over the first sequence. The code (shown here only as illustration) is

```
def scanLeft[A, B](xs: Seq[A])(b0: B)(next: (B, A) => B): Seq[B] = {
  val init: (B, Seq[B]) = (b0, Seq(b0))
  val (_, result) = xs.foldLeft(init){ case ((b, seq), x) =>
    val newB = next(b, x)
    (newB, seq ++ Seq(newB))
  }
  result
}
```

To define the (nameless) updater function for `.foldLeft`, we used the Scala feature that makes it easier to define functions with several arguments containing tuples. In our case, the updater function in `.foldLeft` has two arguments: the first is a tuple `(B, Seq[B])`, the second is a value of type `A`. The pattern matching expression `{ case ((b, seq), x) => ... }` appears to match against a nested tuple. In reality, this expression matches the two arguments of the updater function and, at the same time, destructures the tuple argument as `(b, seq)`.

2.5 Summary

We have seen a broad overview of translating mathematical induction into Scala code. What problems can we solve now?

- Compute mathematical expressions involving arbitrary recursion.
- Use the accumulator trick to enforce tail recursion.
- Use arbitrary inductive (i.e. recursive) formulas to:
  - convert sequences to single values ("aggregation");
  - create new sequences from single values;
  - transform existing sequences into new sequences.

Table 2.1 summarizes Scala methods implementing those tasks.

Using these methods, any iterative calculation is implemented by translating mathematical induction directly into code. In the functional programming paradigm, the programmer does not need to write any loops or check any array indices. Instead, the programmer reasons about sequences as mathematical values: "Starting from this value, we get that sequence, then transform it into this other sequence," etc. This is a powerful way of working with sequences, dictionaries, and sets.

Definition via mathematical induction	Scala code example
$f([]) = b\,;\ \ f(s{+}{+}[x]) = g(f(s), x)$	`f(xs) = xs.foldLeft(b)(g)`
$x_0 = b\,;\ \ x_{k+1} = g(x_k)$	`xs = Stream.iterate(b)(g)`
$y_0 = b\,;\ \ y_{k+1} = g(y_k, x_k)$	`ys = xs.scanLeft(b)(g)`

Table 2.1: Implementing mathematical induction.

Many kinds of programming errors (such as an incorrect array index) are avoided from the outset, and the code is shorter and easier to read than conventional code written using loops.

What tasks are not possible with these tools? We cannot implement a non-tail-recursive function without stack overflow (i.e. without unlimited growth of intermediate expressions). The accumulator trick does not always work! In some cases, it is impossible to implement tail recursion in a given recursive computation. An example of such a computation is the "merge-sort" algorithm where the function body must contain two recursive calls within a single expression. (It is impossible to rewrite *two* recursive calls as one!)

What if our recursive code cannot be transformed into tail-recursive code via the accumulator trick, but the depth of the recursion is so large that stack overflows are possible? We must then use more advanced tricks (for instance, the "continuation-passing" or "trampolines") that convert non-tail-recursive code into iterative code without stack overflows. These tricks are beyond the scope of this chapter.

2.5.1 Solved examples

Example 2.5.1.1 Compute the smallest n such that $f(f(f(...f(1)...) \geq 1000$, where the function f is applied n times. Write this as a function taking f, 1, and 1000 as arguments. Test with $f(x) = 2x + 1$.

Solution: We define a stream of values $[1, f(1), f(f(1)), ...]$ and use `.takeWhile` to stop the stream when the given condition holds. The `.length` method then gives the length of the resulting sequence:

```
scala> Stream.iterate(1)(x => 2*x+1).takeWhile(x => x < 1000).toList
res0: List[Int] = List(1, 3, 7, 15, 31, 63, 127, 255, 511)

scala> Stream.iterate(1)(x => 2*x+1).takeWhile(x => x < 1000).length
res1: Int = 9
```

Example 2.5.1.2 **(a)** For a given stream of integers, compute the stream of the largest values seen so far.

(b) Compute the stream of k largest values seen so far (k is a given integer parameter).

Solution: We cannot use `.max` or sort the entire stream, since the length of the stream is not known in advance. So we need to use `.scanLeft`, which will build the output stream one element at a time.

(a) Maintain the largest value seen so far in the accumulator of the `.scanLeft`:

```
def maxSoFar(xs: Stream[Int]): Stream[Int] =
  xs.scanLeft(xs.head){ case (max, x) => math.max(max, x) }
    .drop(1)
```

We use `.drop(1)` to remove the initial value, `xs.head`, because it is not useful for our result but is necessary for the definition of `.scanLeft`.

To test this function, let us define a stream whose values go up and down:

```
scala> val s = Stream.iterate(0)(x => 1 - 2*x)
s: Stream[Int] = Stream(0, ?)
```

```
scala> s.take(10).toList
res0: List[Int] = List(0, 1, -1, 3, -5, 11, -21, 43, -85, 171)

scala> maxSoFar(s).take(10).toList
res1: List[Int] = List(0, 1, 1, 3, 3, 11, 11, 43, 43, 171)
```

(b) We again use `.scanLeft`, where now the accumulator needs to keep the largest k values seen so far. There are two ways of maintaining this accumulator: First, to have a sequence of k values that we sort and truncate each time. Second, to use a specialized data structure such as a priority queue that automatically keeps values sorted and its length bounded. For the purposes of this tutorial, let us avoid using specialized data structures:

```
def maxKSoFar(xs: Stream[Int], k: Int): Stream[Seq[Int]] = {
  // The initial value of the accumulator is an empty Seq() of type Seq[Int].
  xs.scanLeft(Seq[Int]()) { case (seq, x) =>
  // Sort in the descending order, and take the first k values.
    (seq :+ x).sorted.reverse.take(k)
  }.drop(1) // Skip the useless first value.
}

scala> maxKSoFar(s, 3).take(10).toList
res2: List[Seq[Int]] = List(List(0), List(1, 0), List(1, 0, -1), List(3, 1, 0), List(3, 1, 0),
     List(11, 3, 1), List(11, 3, 1), List(43, 11, 3), List(43, 11, 3), List(171, 43, 11))
```

Example 2.5.1.3 Find the last element of a non-empty sequence. (Use `.reduce`.)

Solution: This function is available in the Scala library as the standard method `.last` on sequences. Here we need to re-implement it using `.reduce`. Begin by writing an inductive definition:

- (Base case.) `last(Seq(x)) = x`.
- (Inductive step.) `last(Seq(x) ++ xs) = last(xs)` assuming `xs` is non-empty.

The `.reduce` method implements an inductive aggregation similarly to `.foldLeft`, except that for `.reduce` the base case is fixed – it always returns `x` for a 1-element sequence `Seq(x)`. This is exactly what we need here, so the inductive definition is directly translated into code, with the updater function $g(x, y) = y$:

```
def last[A](xs: Seq[A]): A = xs.reduce { case (x, y) => y }
```

Example 2.5.1.4 **(a)** Using tail recursion, implement the binary search algorithm in a given sorted sequence `xs: Seq[Int]` as a function returning the index of the requested number `n` (assume that `xs` contains the number `n`):

```
def binSearch(xs: Seq[Int], goal: Int): Int = ???

scala> binSearch(Seq(1, 3, 5, 7), 5)
res0: Int = 2
```

(b) Re-implement `binSearch` using `Stream.iterate` instead of explicit recursion.

Solution: **(a)** The well-known binary search algorithm splits the array into two halves and may continue the search recursively in one of the halves. We need to write the solution as a tail-recursive function with an additional accumulator argument. So we expect that the code should look like this,

```
def binSearch(xs: Seq[Int], goal: Int, acc: _ = ???): Int = {
  if (???are we done???) acc
  else {
    // Determine which half of the sequence contains ‘goal‘.
    // Then update the accumulator accordingly.
    val newAcc = ???
```

```
    binSearch(xs, goal, newAcc) // Tail-recursive call.
  }
}
```

It remains to determine the type and the initial value of the accumulator, as well as the code for updating it.

The information required for the recursive call is the remaining segment of the sequence where the target number is present. This segment is defined by two indices i, j representing the left and the right bounds of the sub-sequence, such that the target element is x_n with $x_i \leq x_n < x_{j-1}$. It follows that the accumulator should be a pair of two integers (i, j). The initial value of the accumulator is the pair $(0, N)$ where N is the length of the entire sequence. The search is finished when $i + 1 = j$. We can now write the corresponding code, where for convenience we introduce *two* accumulator values:

```
@tailrec def binSearch(xs: Seq[Int], goal: Int)(left: Int = 0,
                       right: Int = xs.length): Int = {
  // Check whether ‘goal‘ is at one of the boundaries.
  if (right - left <= 1 || xs(left) == goal) left
  else {
    val middle = (left + right) / 2
    // Determine which half of the array contains ‘target‘.
    // Update the accumulator accordingly.
    val (newLeft, newRight) =
      if (goal < xs(middle)) (left, middle)
      else (middle, right)
    binSearch(xs, goal)(newLeft, newRight) // Tail-recursive call.
  }
}

scala> binSearch(0 to 10, 3)() // Default accumulator values.
res0: Int = 3
```

Here we used a feature of Scala that allows us to use `xs.length` as a default value for the argument `right` of `binSearch`. This is possible only because `right` is in a different **argument group** from `xs`. In Scala, values in an argument group may depend on arguments given in a *previous* argument group. However, the code

```
def binSearch(xs: Seq[Int], goal: Int, left: Int = 0, right: Int = xs.length)
```

will generate an error: the arguments in the same argument group cannot depend on each other. (The error will say `not found: value xs`.)

(b) We can visualize the binary search as a procedure that generates a sequence of progressively tighter bounds for the location of `goal`. The initial bounds are `(0, xs.length)`, and the final bounds are `(k, k+1)` for some `k`. We can generate the sequence of bounds using `Stream.iterate` and stop the sequence when the bounds become sufficiently tight. To make the use of `.takeWhile` more convenient, we add an extra sequence element where the bounds `(k, k)` are equal. The code becomes

```
def binSearch(xs: Seq[Int], goal: Int): Int = {
  type Acc = (Int, Int)
  val init: Acc = (0, xs.length)
  val updater: Acc => Acc = { case (left, right) =>
    if (right - left <= 1) (left, left) // Extra element.
    else if (xs(left) == goal) (left, left + 1)
    else {
      val middle = (left + right) / 2
      // Determine which half of the array contains ‘target‘.
      // Update the accumulator accordingly.
      if (goal < xs(middle)) (left, middle)
      else (middle, right)
    }
```

```
  }
  Stream.iterate(init)(updater)
    .takeWhile{ case (left, right) => right > left }
    .last._1 // Take the 'left' boundary from the last element.
}
```

This code is clearer because recursion is delegated to `Stream.iterate`, and we only need to write the "business logic" (i.e. the base case and the inductive step) of our function.

Example 2.5.1.5 For a given positive `n:Int`, compute the sequence $[s_0, s_1, s_2, ...]$ defined by $s_0 = SD(n)$ and $s_k = SD(s_{k-1})$ for $k > 0$, where $SD(x)$ is the sum of the decimal digits of the integer x, e.g. $SD(123) = 6$. Stop the sequence s_i when the numbers begin repeating. For example, $SD(99) = 18$, $SD(18) = 9$, $SD(9) = 9$. So, for $n = 99$, the sequence s_i must be computed as $[99, 18, 9]$.

Hint: use `Stream.iterate`; compute the decimal digits in the reverse order since the sum will be the same.

Solution: We need to implement a function `sdSeq` having the type signature

```
def sdSeq(n: Int): Seq[Int]
```

First we need to implement $SD(x)$. The sum of digits is obtained by almost the same code as in Section 2.3:

```
def SD(n: Int): Int = if (n == 0) 0 else
  Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ % 10).sum
```

Now we can try evaluating SD on some numbers to see its behavior:

```
scala> (1 to 15).toList.map(SD)
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6)
```

It is clear that $SD(n) < n$ as long as $n \geq 10$. So the sequence elements s_i will not repeat until they become smaller than 10, and then they will always repeat. This seems to be an easy way of stopping the sequence. Let us try that:

```
scala> Stream.iterate(99)(SD).takeWhile(x => x >= 10).toList
res1: List[Int] = List(99, 18)
```

We are missing the last element of the sequence, $SD(18) = 9$, because `.takeWhile` stops the sequence too early. In order to obtain the correct sequence, we need to compute one more element. To fix this, we can generate a stream of *pairs*:

```
scala> Stream.iterate((0, 99)){ case (prev, x) => (x, SD(x)) }.
  takeWhile{ case (prev, x) => prev >= 10 || x >= 10 }.toList
res2: List[(Int, Int)] = List((0,99), (99,18), (18,9))
```

This looks right; it remains to remove the first parts of the tuples:

```
def sdSeq(n: Int): Seq[Int] =
  Stream.iterate((0, n)){ case (prev, x) => (x, SD(x)) } // Stream[(Int, Int)]
    .takeWhile{ case (prev, x) => prev >= 10 || x >= 10 } // Stream[(Int, Int)]
    .map(_._2) // Stream[Int]
    .toList // List[Int]

scala> sdSeq(99)
res3: Seq[Int] = List(99, 18, 9)
```

Example 2.5.1.6 For a given stream $[s_0, s_1, s_2, ...]$ of type `Stream[T]`, compute the "half-speed" stream $h = [s_0, s_0, s_1, s_1, s_2, s_2, ...]$. (The half-speed sequence h can be defined by the formula $s_k = h_{2k} = h_{2k+1}$.)

Solution: We use `.map` to replace each element s_i by a sequence containing two copies of s_i. Let us try this on a sample sequence:

```
scala> Seq(1,2,3).map( x => Seq(x, x))
res0: Seq[Seq[Int]] = List(List(1, 1), List(2, 2), List(3, 3))
```

The result is almost what we need, except we need to `.flatten` the nested list:

```
scala> Seq(1,2,3).map( x => Seq(x, x)).flatten
res1: Seq[Seq[Int]] = List(1, 1, 2, 2, 3, 3)
```

The composition of `.map` and `.flatten` is `.flatMap`, so the final code is

```
def halfSpeed[T](str: Stream[T]): Stream[T] = str.flatMap(x => Seq(x, x))

scala> halfSpeed(Seq(1,2,3).toStream)
res2: Stream[Int] = Stream(1, ?)

scala> halfSpeed(Seq(1,2,3).toStream).toList
res3: List[Int] = List(1, 1, 2, 2, 3, 3)
```

Example 2.5.1.7 Stop a given stream $[s_0, s_1, s_2, ...]$ at a place k where the sequence repeats itself; that is, an element s_k equals some earlier element s_i with $i < k$.

Solution: The trick is to create a half-speed sequence h_i out of s_i and then find an index $k > 0$ such that $h_k = s_k$. (The condition $k > 0$ is needed because we will always have $h_0 = s_0$.) If we find such an index k, it would mean that either $s_k = s_{k/2}$ or $s_k = s_{(k-1)/2}$; in either case, we will have found an element s_k that equals an earlier element.

As an example, take $s = [1, 3, 5, 7, 9, 3, 5, 7, 9, ...]$ and compute the half-speed sequence $h = [1, 1, 3, 3, 5, 5, 7, 7, 9, 9, 3$ Looking for an index $k > 0$ such that $h_k = s_k$, we find that $s_7 = h_7 = 7$. This is indeed an element of s_i that repeats an earlier element (although s_7 is not the first such repetition).

There are in principle two ways of finding an index $k > 0$ such that $h_k = s_k$: First, to iterate over a list of indices $k = 1, 2, ...$ and evaluate the condition $h_k = s_k$ as a function of k. Second, to build a sequence of pairs (h_i, s_i) and use `.takeWhile` to stop at the required index. In the present case, we cannot use the first way because we do not have a fixed set of indices to iterate over. Also, the condition $h_k = s_k$ cannot be directly evaluated as a function of k because s and h are streams that compute elements on demand, not lists whose elements are computed in advance and ready to be used.

So the code must iterate over a stream of pairs (h_i, s_i):

```
def stopRepeats[T](str: Stream[T]): Stream[T] = {
  val halfSpeed = str.flatMap(x => Seq(x, x))
  val result = halfSpeed.zip(str) // Stream[(T, T)]
  .drop(1) // Enforce the condition k > 0.
  .takeWhile { case (h, s) => h != s } // Stream[(T, T)]
  .map(_._2) // Stream[T]
  str.head +: result // Prepend the first element that was dropped.
}

scala> stopRepeats(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toStream).toList
res0: List[Int] = List(1, 3, 5, 7, 9, 3, 5)
```

Example 2.5.1.8 Reverse each word in a string, but keep the order of words:

```
def revWords(s: String): String = ???

scala> revWords("A quick brown fox")
res0: String = A kciuq nworb xof
```

Solution: The standard method `.split` converts a string into an array of words:

```
scala> "pa re ci vo mu".split(" ")
res0: Array[String] = Array(pa, re, ci, vo, mu)
```

Each word is reversed with `.reverse`; the resulting array is concatenated into a string with `.mkString`. So the code is

```
def revWords(s: String): String = s.split(" ").map(_.reverse).mkString(" ")
```

Example 2.5.1.9 Remove adjacent repeated characters from a string:

```
def noDups(s: String): String = ???

scala> noDups("abbcdeeeeefddgggggh")
res0: String = abcdefdgh
```

Solution: A string is automatically converted into a sequence of characters when we use methods such as `.map` or `.zip` on it. So, we can use `s.zip(s.tail)` to get a sequence of pairs (s_k, s_{k+1}) where c_k is the k-th character of the string s. Now we can use `.filter` to remove the elements s_k for which $s_{k+1} = s_k$:

```
scala> val s = "abbcd"
s: String = abbcd

scala> s.zip(s.tail).filter { case (sk, skPlus1) => sk != skPlus1 }
res0: IndexedSeq[(Char, Char)] = Vector((a,b), (b,c), (c,d))
```

It remains to convert this sequence of pairs into the string `"abcd"`. One way of doing this is to project the sequence of pairs onto the second parts of the pairs,

```
scala> res0.map(_._2).mkString
res1: String = bcd
```

We just need to add the first character, `'a'`. The resulting code is

```
def noDups(s: String): String = if (s == "") "" else {
  val pairs = s.zip(s.tail).filter { case (x, y) => x != y }
  pairs.head._1 +: pairs.map(_._2).mkString
}
```

The method `+:` prepends an element to a sequence, so `x +: xs` is equivalent to `Seq(x) ++ xs`.

Example 2.5.1.10 **(a)** Count the occurrences of each distinct word in a string:

```
def countWords(s: String): Map[String, Int] = ???

scala> countWords("a quick a quick a fox")
res0: Map[String, Int] = Map("a" -> 3, "quick" -> 2, "fox" -> 1)
```

(b) Count the occurrences of each distinct element in a sequence of type `Seq[A]`.

Solution: **(a)** We split the string into an array of words via `s.split(" ")`, and apply a `.foldLeft` to that array, since the computation is a kind of aggregation over the array of words. The accumulator of the aggregation will be the dictionary of word counts for all the words seen so far:

```
def countWords(s: String): Map[String, Int] = {
  val init: Map[String, Int] = Map()
  s.split(" ").foldLeft(init) { (dict, word) =>
    val newCount = dict.getOrElse(word, 0) + 1
    dict.updated(word, newCount)
  }
```

```
}
```

(b) The main code of `countWords` does not depend on the fact that words are of type `String`. It will work in the same way for any other type of keys for the dictionary. So we keep the same code and define the type signature of the function to contain a type parameter `A` instead of `String`:

```
def countValues[A](xs: Seq[A]): Map[A, Int] =
  xs.foldLeft(Map[A, Int]()) { (dict, word) =>
    val newCount = dict.getOrElse(word, 0) + 1
    dict.updated(word, newCount)
  }

scala> countValues(Seq(100, 100, 200, 100, 200, 200, 100))
res0: Map[Int,Int] = Map(100 -> 4, 200 -> 3)
```

Example 2.5.1.11 For a given sequence of type `Seq[A]`, find the longest subsequence that does not contain any adjacent duplicate values.

```
def longestNoDups[A](xs: Seq[A]): Seq[A] = ???

scala> longestNoDups(Seq(1, 2, 2, 5, 4, 4, 4, 8, 2, 3, 3))
res0: Seq[Int] = List(4, 8, 2, 3)
```

Solution: This is a dynamic programming problem. Many such problems are solved with a single `.foldLeft`. The accumulator represents the current "state" of the dynamic programming solution, and the "state" is updated with each new element of the input sequence.

To obtain the solution, we first need to determine the type of the accumulator value, or the "state". The task is to find the longest subsequence without adjacent duplicates. So the accumulator should represent the longest subsequence found so far, as well as any required extra information about other subsequences that might grow as we iterate over the elements of `xs`. What is this extra information in our case?

Imagine that we wanted to build set of *all* subsequences without adjacent duplicates. In the example where the input sequence is $[1,2,2,5,4,4,4,8,2,3,3]$, this set of subsequences should be $\{[1,2],[2,5,4],[4,8,2,3]\}$. We can build this set incrementally in the accumulator value of a `.foldLeft`. To visualize how this set would be built, consider the partial result after seeing the first 8 elements of the input sequence, $[1,2,2,5,4,4,4,8]$. The partial set of non-repeating subsequences is $\{[1,2],[2,5,4],[4,8]\}$. As we add another element, 2, we update the partial set to $\{[1,2],[2,5,4],[4,8,2]\}$.

It is now clear that the subsequence $[1,2]$ has no chance of being the longest subsequence, since $[2,5,4]$ is already longer. However, we do not yet know whether $[2,5,4]$ or $[4,8,2]$ is the winner, because the subsequence $[4,8,2]$ could still grow and become the longest one (and it does become $[4,8,2,3]$ later). At this point, we need to keep both of these two subsequences in the accumulator, but we may already discard $[1,2]$.

We have deduced that the accumulator needs to keep only *two* sequences: the first sequence is already terminated and will not grow, the second sequence ends with the current element and may yet grow. The initial value of the accumulator is empty. The first subsequence is discarded when it becomes shorter than the second. The code can be written now:

```
def longestNoDups[A](xs: Seq[A]): Seq[A] = {
  val init: (Seq[A], Seq[A]) = (Seq(), Seq())
  val (first, last) = xs.foldLeft(init) { case ((first, current), x) =>
    // If ‘current‘ is empty, ‘x‘ cannot be repeated.
    val xWasRepeated = current != Seq() && current.last == x
    val firstIsLongerThanCurrent = first.length > current.length
    // Compute the new pair ‘(first, current)‘.
    // Keep ‘first‘ only if it is longer; otherwise replace it by ‘current‘.
    val newFirst = if (firstIsLongerThanCurrent) first else current
    // Append ‘x‘ to ‘current‘ if ‘x‘ is not repeated.
```

```
    val newCurrent = if (xWasRepeated) Seq(x) else current :+ x
    (newFirst, newCurrent)
  }
  // Return the longer of the two subsequences; prefer ‘first‘.
  if (first.length >= last.length) first else last
}
```

2.5.2 Exercises

Exercise 2.5.2.1 Compute the sum of squared digits of a given integer; e.g., `dsq(123) = 14` (see Example 2.5.1.5). Generalize the solution to take an arbitrary function `f : Int => Int` as a parameter, instead of the squaring operation. The type signature and a sample test:

```
def digitsMapSum(x: Int)(f: Int => Int): Int = ???

scala> digitsMap(123){ x => x * x }
res0: Int = 14

scala> digitsMap(123){ x => x * x * x }
res1: Int = 36
```

Exercise 2.5.2.2 Compute the **Collatz sequence** c_i as a stream defined by

$$c_0 = n \quad ; \qquad c_{k+1} = \begin{cases} \frac{c_k}{2} & \text{if } c_k \text{ is even,} \\ 3c_k + 1 & \text{if } c_k \text{ is odd.} \end{cases}$$

Stop the stream when it reaches 1 (as one would expect it will).

Exercise 2.5.2.3 For a given integer n, compute the sum of cubed digits, then the sum of cubed digits of the result, etc.; stop the resulting sequence when it repeats itself, and so determine whether it ever reaches 1. (Use Exercise 2.5.2.1.)

```
def cubes(n: Int): Stream[Int] = ???

scala> cubes(123).take(10).toList
res0: List[Int] = List(123, 36, 243, 99, 1458, 702, 351, 153, 153, 153)

scala> cubes(2).take(10).toList
res1: List[Int] = List(2, 8, 512, 134, 92, 737, 713, 371, 371, 371)

scala> cubes(4).take(10).toList
res2: List[Int] = List(4, 64, 280, 520, 133, 55, 250, 133, 55, 250)

def cubesReach1(n: Int): Boolean = ???

scala> cubesReach1(10)
res3: Boolean = true

scala> cubesReach1(4)
res4: Boolean = false
```

Exercise 2.5.2.4 For `a`, `b`, `c` of type `Set[Int]`, compute the set of all sets of the form `Set(x, y, z)` where `x` is from `a`, `y` from `b`, and `z` from `c`. The required type signature and a sample test:

```
def prod3(a: Set[Int], b: Set[Int], c: Set[Int]): Set[Set[Int]] = ???

scala> prod3(Set(1,2), Set(3), Set(4,5))
res0: Set[Set[Int]] = Set(Set(1,3,4), Set(1,3,5), Set(2,3,4), Set(2,3,5))
```

Hint: use `.flatMap`.

Exercise 2.5.2.5* Same task as in Exercise 2.5.2.4 for a set of sets, i.e. given a `Set[Set[Int]]` instead of just three sets `a`, `b`, `c`. The required type signature and a sample test:

```
def prodSet(si: Set[Set[Int]]): Set[Set[Int]] = ???

scala> prodSet(Set(Set(1,2), Set(3), Set(4,5), Set(6)))
res0: Set[Set[Int]] = Set(Set(1,3,4,6),Set(1,3,5,6),Set(2,3,4,6),Set(2,3,5,6))
```

Hint: use `.foldLeft` and `.flatMap`.

Exercise 2.5.2.6* In a sorted array `xs:Array[Int]` where no values are repeated, find all pairs of values whose sum equals a given number n. Use tail recursion. A possible type signature and a sample test:

```
def pairs(goal: Int, xs: Array[Int]): Set[(Int, Int)] = ???

scala> pairs(10, Array(1, 2, 3, 4, 5, 6, 7, 8))()
res0: Set[(Int, Int)] = Set((2,8), (3,7), (4,6), (5,5))
```

Exercise 2.5.2.7 Reverse a sentence's word order, but keep the words unchanged:

```
def revSentence(s: String): String = ???

scala> revSentence("A quick brown fox")
res0: String = "fox brown quick A"
```

Exercise 2.5.2.8 Reverse an integer's digits (see Example 2.5.1.5) as shown:

```
def revDigits(n: Int): Int = ???

scala> revDigits(12345)
res0: Int = 54321
```

A **palindrome number** is an integer `n` such that `revDigits(n) == n`. Write a function `Int => Boolean` that checks whether a given positive integer is a palindrome.

Exercise 2.5.2.9 Starting from a given integer `n`, compute `revDigits(n) + n`; the function `revDigits` was defined in Exercise 2.5.2.8. Check whether the result is a palindrome integer. If it is not, repeat the same operation until a palindrome number is found, and return that number. The required type signature and a test:

```
def findPalindrome(n: Int): Int = ???

scala> findPalindrome(123)
res0: Int = 444

scala> findPalindrome(83951)
res1: Int = 869363968
```

Exercise 2.5.2.10 **(a)** For a given integer interval $[n_1, n_2]$, find the largest integer $k \in [n_1, n_2]$ such that the decimal representation of k does *not* contain any of the digits 3, 5, or 7. **(b)** For a given integer interval $[n_1, n_2]$, find the integer $k \in [n_1, n_2]$ with the largest sum of decimal digits. **(c)** A positive integer n is called a **perfect number** if it is equal to the sum of its divisors (other integers k such that $k < n$ and n/k is an integer). For example, 6 is a perfect number because its divisors are 1, 2, and 3, and $1 + 2 + 3 = 6$, while 8 is not a perfect number because its divisors are 1, 2, and 4, and $1 + 2 + 4 = 7 \neq 8$. Write a function that determines whether a given number n is perfect. Determine all perfect numbers up to one million.

Exercise 2.5.2.11 Remove adjacent repeated elements from a sequence of type `Seq[A]` when they are repeated more than k times. Repetitions up to k times should remain unchanged. The required type signature and a sample test:

```
def removeDups[A](s: Seq[A], k: Int): Seq[A] = ???

scala> removeDups(Seq(1, 1, 1, 1, 5, 2, 2, 5, 5, 5, 5, 5, 1), 3)
res0: Seq[Int] = List(1, 1, 1, 5, 2, 2, 5, 5, 5, 1)
```

Exercise 2.5.2.12 **(a)** Remove repeated elements (whether adjacent or not) from a sequence of type `Seq[A]`. (This re-implements the standard method `.distinct`.)

(b) For a sequence of type `Seq[A]`, remove all elements that are repeated (whether adjacent or not) more than k times:

```
def removeK[A](k: Int, xs: Seq[A]): Seq[A] = ???

scala> removeK(2, Seq("a", "b", "a", "b", "b", "c", "b", "a"))
res0: Seq[String] = List(a, b, a, b, c)
```

Exercise 2.5.2.13* For a given sequence `xs:Seq[Double]`, find a subsequence that has the largest sum of values. The sequence `xs` is not sorted, and its values may be positive or negative. The required type signature and a sample test:

```
def maxsub(xs: Seq[Double]): Seq[Double] = ???

scala> maxsub(Seq(1.0, -1.5, 2.0, 3.0, -0.5, 2.0, 1.0, -10.0, 2.0))
res0: Seq[Double] = List(2.0, 3.0, -0.5, 2.0, 1.0)
```

Hint: use dynamic programming and `.foldLeft`.

Exercise 2.5.2.14* Find all common integers between two sorted sequences:

```
def commonInt(xs: Seq[Int], ys: Seq[Int]): Seq[Int] = ??? // Use tail recursion.

scala> commonInt(Seq(1, 3, 5, 7), Seq(2, 3, 4, 6, 7, 8))
res0: Seq[Int] = List(3, 7)
```

2.6 Discussion

2.6.1 Total and partial functions

In Scala, functions can be total or partial. A **total** function will always compute a result value, while a **partial** function may fail to compute its result for certain values of its arguments.

A simple example of a partial function in Scala is the `.max` method: it only works for non-empty sequences. Trying to evaluate it on an empty sequence generates an error called an "exception":

```
scala> Seq(1).tail
res0: Seq[Int] = List()
scala> res0.max
java.lang.UnsupportedOperationException: empty.max
  at scala.collection.TraversableOnce$class.max(TraversableOnce.scala:229)
  at scala.collection.AbstractTraversable.max(Traversable.scala:104)
  ... 32 elided
```

This kind of error may crash the entire program at run time. Unlike the type errors we saw before, which occur at compilation time (i.e. before the program can start), **run-time errors** occur while the program is running, and only when some partial function happens to get an incorrect input. The

incorrect input may occur at any point after the program started running, which may crash the entire program in the middle of a long computation.

So, it seems clear that we should write code that does not generate such errors. For instance, it is safe to apply `.max` to a sequence if we know that it is non-empty.

Sometimes, a function that uses pattern matching turns out to be a partial function because its pattern matching code fails on certain input data.

If a pattern matching expression fails, the code will throw an exception and stop running. In functional programming, we usually want to avoid this situation because it makes it much harder to reason about program correctness. In most cases, programs can be written to avoid the possibility of match errors. An example of an unsafe pattern matching expression is

```
def h(p: (Int, Int)): Int = p match { case (x, 0) => x }

scala> h( (1,0) )
res0: Int = 1

scala> h( (1,2) )
 scala.MatchError: (1,2) (of class scala.Tuple2$mcII$sp)
  at .h(<console>:12)
  ... 32 elided
```

Here the pattern contains a pattern variable `x` and a constant `0`. This pattern only matches tuples whose second part is equal to `0`. If the second argument is nonzero, a match error occurs and the program crashes. So, `h` is a partial function.

Pattern matching failures never happen if we match a tuple of correct size with a pattern such as `(x, y, z)`, because pattern variables will always match whatever values the tuple has. So, pattern matching with a pattern such as `(x, y, z)` is **infallible** (never fails at run time) when applied to a tuple with 3 elements.

Another way in which pattern matching can be made infallible is by including a pattern that matches everything:

```
p match {
  case (x, 0) => ... // This only matches some tuples.
  case _ => ... // This matches everything.
}
```

If the first pattern `(x, 0)` fails to match the value of `p`, the second pattern will be tried (and will always succeed). When a `match` expression has several `case` patterns, the patterns are tried in the order they are written. So, a match expression can be made infallible by adding a "match-all" underscore pattern.

2.6.2 Scope and shadowing of pattern matching variables

Pattern matching introduces **locally scoped** variables – that is, variables defined only on the right-hand side of the pattern match expression. As an example, consider this code:

```
def f(x: (Int, Int)): Int = x match { case (x, y) => x + y }

scala> f( (2,4) )
res0: Int = 6
```

The argument of `f` is the variable `x` of a tuple type `(Int,Int)`, but there is also a pattern variable `x` in the case expression. The pattern variable `x` matches the first part of the tuple and has type `Int`. Because variables are locally scoped, the pattern variable `x` is only defined within the expression `x + y`. The argument `x:(Int,Int)` is a completely different variable whose value has a different type.

The code works correctly but is confusing to read because of the name clash between the two

quite different variables, both named `x`. Another negative consequence of the name clash is that the argument `x:(Int,Int)` *is invisible* within the case expression: if we write "`x`" in that expression, we will get the pattern variable `x:Int`. One says that the argument `x:(Int,Int)` has been **shadowed** by the pattern variable `x`.

The problem is easy to correct: we can give the pattern variable some other name. Since the pattern variable is locally scoped, it can be renamed within its scope without having to change any other code. A completely equivalent code is

```
def f(x: (Int, Int)): Int = x match { case (a, b) => a + b }

scala> f( (2,4) )
res0: Int = 6
```

2.6.3 Lazy values and sequences: Iterators and streams

We have used streams to create sequences whose length is not known in advance. An example is a stream containing a sequence of increasing positive integers:

```
scala> val p = Stream.iterate(1)(_ + 1)
p: Stream[Int] = Stream(1, ?)
```

At this point, we have not defined a stopping condition for this stream. In some sense, streams are "infinite" sequences, although in practice a stream is always finite because computers cannot run infinitely long. Also, computers cannot store infinitely many values in memory.

To be more precise, streams are "not fully computed" rather than "infinite". The main difference between arrays and streams is that a stream's elements are computed on demand and not initially available (except perhaps for the first element), while an array's elements are all computed in advance and are available immediately. Generally, there are four possible ways a value could be available:

Availability	Explanation	Example Scala code
"eager"	computed in advance	`val x = f(123)`
"lazy"	computed upon first request	`lazy val y = f(123)`
"on-call"	computed each time it is requested	`def z = f(123)`
"never"	cannot be computed due to errors	`val (x, y) = "abc"`

A **lazy value** (declared as `lazy val` in Scala) is computed only when used in some other expression. Once computed, a lazy value stays in memory and will not be re-computed.

An "on-call" value is re-computed every time it is used. In Scala, this is the behavior of a `def` declaration.

Most collection types in Scala (such as `List`, `Array`, `Set`, and `Map`) are **eager**: all the elements inside these collections are already evaluated. A stream can be seen as a **lazy collection**. Values in a stream are computed only when first needed; after that, they remain in memory and will not be computed again:

```
scala> val str = Stream.iterate(1)(_ + 1)
str: Stream[Int] = Stream(1, ?)

scala> str.take(10).toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> str
res1: Stream[Int] = Stream(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ?)
```

In many cases, it is not necessary to keep previous values of a sequence in memory. For example, consider the computation

```
scala> (1L to 1000000000L).sum
res0: Long = 500000000500000000
```

We do not actually need to keep a billion numbers in memory if we only want to compute their sum. Indeed, the computation just shown does *not* keep all the numbers in memory. The same computation fails if we use a list or a stream:

```
scala> (1L to 1000000000L).toStream.sum
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

The code `(1L to 1000000000L).sum` works because the operation `(1 to n)` produces a sequence whose elements are computed whenever needed but do not remain in memory. This can be seen as a sequence with the "on-call" availability of elements. Sequences of this sort are called **iterators**. Here are some examples:

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 1 until 5
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

The types `Range` and `Range.Inclusive` are defined in the Scala standard library and are iterators. They behave as collections and support the usual methods (`.map`, `.filter`, etc.), but they do not store previously computed values in memory.

The `.view` method Eager collections such as `List` or `Array` can be converted to iterators by using the `.view` method. This is necessary when intermediate collections consume too much memory when fully evaluated. For example, consider the computation of Example 2.1.5.7 where we used `.flatMap` to replace each element of an initial sequence by three new numbers before computing `.max` of the resulting collection. If instead of three new numbers we wanted to compute *three million* new numbers each time, the intermediate collection created by `.flatMap` would require too much memory, and the computation would crash:

```
scala> (1 to 10).flatMap(x => 1 to 3000000).max
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Even though the range expression `(1 to 10)` produces an iterator, a subsequent `.flatMap` operation creates an intermediate collection that is too large for our computer's memory. We can use `.view` to avoid this:

```
scala> (1 to 10).view.flatMap(x => 1 to 3000000).max
res0: Int = 3000000
```

The choice between using streams and using iterators is dictated by the memory considerations. Except for that, streams and iterators behave similarly to other sequences. We may write programs in the map/reduce style, applying the standard methods such as `.map`, `.filter`, etc., to streams and iterators. Mathematical reasoning about sequences is the same, whether they are eager, lazy, or on-call.

The broken `Iterator` class The Scala library contains a class called `Iterator`, which has methods such as `Iterator.iterate` and other methods similar to `Stream`. However, `Iterator` actually not an "iterator" in the sense I explained. It cannot be treated as a *value* in the mathematical sense:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator
```

```
scala> iter.toList // Look at the elements of ‘iter‘.
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> iter.toList // Look at these elements again...??
res1: List[Int] = List()

scala> iter
res2: Iterator[Int] = empty iterator
```

Evaluating the expression `iter.toList` two times produces a *different* result the second time! Also, we see that `iter` became "empty" after the first use.

This situation is impossible in mathematics: if x is some value, such as 100, and f is some function, such as $f(x) = \sqrt{x}$, then $f(x)$ will be the same, $f(100) = \sqrt{100} = 10$, no matter how many times we compute $f(x)$. For instance, we can compute $f(x) + f(x) = 20$ and obtain the correct result. The number $x = 100$ does not "become empty" after the first use; its value remains the same. This behavior is called the **value semantics** of numbers. One says that integers "are values" in the mathematical sense. Alternatively, one says that numbers are **immutable,** i.e. cannot be changed. (What would it mean to "modify" the number 10?)

In programming, a type has value semantics if any computation applied to it always gives the same result. Usually, this means that the type contains immutable data. We can see that Scala's `Range` has value semantics and is immutable:

```
scala> val x = 1 until 10
x: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Collections such as `List`, `Map`, or `Stream` are immutable. Some elements of a `Stream` may not be evaluated yet, but this does not affect its value semantics:

```
scala> val str = (1 until 10).toStream
str: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Iterators produced by applying `.view` also have value semantics:

```
scala> val v = (1 until 10).view
v: scala.collection.SeqView[Int,IndexedSeq[Int]] = SeqView(...)

scala> v.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> v.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Due to the lack of value semantics, programs written using `Iterator` cannot use the tools of mathematical reasoning. This makes it easy to write wrong code that looks correct!

To illustrate the problem, let us re-implement Example 2.5.1.7 by keeping the same code but using `Iterator` instead of `Stream`:

```
def stopRepeatsBad[T](iter: Iterator[T]): Iterator[T] = {
```

```
  val halfSpeed = iter.flatMap(x => Seq(x, x))
  halfSpeed.zip(iter) // Do not prepend the first element. It won't help.
  .drop(1)
  .takeWhile { case (h, s) => h != s }
  .map(_._2)
}

scala> stopRepeatsBad(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toIterator).toList
res0: List[Int] = List(5, 9, 3, 7, 9)
```

The result $[5, 9, 3, 7, 9]$ is incorrect, but not in an obvious way: the sequence *was* stopped at a repetition, as we expected, but some of the elements of the given sequence are missing (while other elements are present). It is difficult to debug a program when it produces numbers that are *partially* correct!

The error in this code occurs in the expression `halfSpeed.zip(iter)` due to the fact that `halfSpeed` was itself defined via `iter`. The result is that `iter` is *used twice* in this code, which leads to errors because `iter` is not immutable and does not behave as a value. Creating an `Iterator` and using it twice in the same expression can even fail with an exception:

```
scala> val s = (1 until 10).toIterator
s: Iterator[Int] = non-empty iterator

scala> val t = s.zip(s).toList
java.util.NoSuchElementException: next on empty iterator
```

It is surprising and counter-intuitive that a variable cannot be used twice! We expect code such as `s.zip(s)` to work correctly even though the variable `s` is used twice. When we read the expression `s.zip(s)`, we imagine a given sequence `s` being "zipped" with itself. So we reason that `s.zip(s)` should produce a sequence of pairs. But Scala's `Iterator` is not immutable, which breaks the usual ways of mathematical reasoning about code.

An `Iterator` can be converted to a `Stream` using the `.toStream` method. This restores the value semantics, since streams are values:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> val str = iter.toStream
str: Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.zip(str).toList
res2: List[(Int, Int)] = List((1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9))
```

Instead of `Iterator`, we can use `Stream` and `.view` when lazy or on-call collections are required.

3 The logic of types. I. Disjunctive types

Disjunctive types describe values that belong to a disjoint set of alternatives.

To see how Scala implements disjunctive types, we need to begin by looking at "case classes".

3.1 Scala's case classes

3.1.1 Tuple types with names

It is often helpful to use names for the different parts of a tuple. Suppose that some program represents the size and the color of socks with the tuple type `(Double, String)`. What if the same tuple type `(Double, String)` is used in another place in the program to mean the amount paid and the payee? A programmer could mix the two values by mistake, and it would be hard to find out why the program incorrectly computes, say, the total amount paid.

```
def totalAmountPaid(ps: Seq[(Double, String)]): Double = ps.map(_._1).sum
val x = (10.5, "white")       // Sock size and color.
val y = (25.0, "restaurant")  // Payment amount and payee.

scala> totalAmountPaid(Seq(x, y)) // Nonsense.
res0: Double = 35.5
```

We would prevent this kind of mistake if we could use two *different* types, with names such as `MySock` and `Payment`, for the two kinds of data. There are three basic ways of defining a new named type in Scala: using a type alias, using a class (or "trait"), and using an opaque type.

Opaque types (hiding a type under a new name) is a feature of a future version of Scala 3; so we focus on type aliases and case classes.

A **type alias** is an alternative name for an existing (already defined) type. We could use type aliases in our example to add clarity to the code:

```
type MySockTuple = (Double, String)
type PaymentTuple = (Double, String)

scala> val s: MySockTuple = (10.5, "white")
s: MySockTuple = (10.5,white)

scala> val p: PaymentTuple = (25.0, "restaurant")
p: PaymentTuple = (25.0,restaurant)
```

But the mix-up error is not prevented:

```
scala> totalAmountPaid(Seq(s, p)) // Nonsense again.
res1: Double = 35.5
```

Scala's **case classes** can be seen as "tuples with names". A case class is equivalent to a tuple type that has a name that we choose when we define the case class. Also, each part of the case class will have a separate name that we must choose. This is how to define case classes for the example with socks and payments:

```
case class MySock(size: Double, color: String)
case class Payment(amount: Double, name: String)
```

```
scala> val sock = MySock(10.5, "white")
sock: MySock = MySock(10.5,white)

scala> val paid = Payment(25.0, "restaurant")
paid: Payment = Payment(25.0,restaurant)                                    ^
```

The new types `MySock` and `Payment` were defined. Values of type `MySock` are written as `MySock(10.5, "white")`, which is similar to writing the tuple `(10.5, "white")` except for adding the name `MySock` in front of the tuple.

To access the parts of a case class, we use the part names:

```
scala> sock.size
res2: Double = 10.5

scala> paid.amount
res3: Double = 25.0
```

The mix-up error is now a type error flagged by the compiler:

```
def totalAmountPaid(ps: Seq[Payment]): Double = ps.map(_.amount).sum

scala> totalAmountPaid(Seq(paid, paid))
res4: Double = 50.0

scala> totalAmountPaid(Seq(sock, paid))
<console>:19: error: type mismatch;
 found   : MySock
 required: Payment
       totalAmountPaid(Seq(sock, paid))
                           ^
```

A function whose argument is of type `MySock` cannot be applied to an argument of type `Payment` or of type `(Double, String)`. Case classes with different names are *different types*, even if they contain the same types of parts.

Just as tuples can have any number of parts, case classes can have any number of parts, but the part names must be distinct, for example:

```
case class Person(firstName: String, lastName: String, age: Int)

scala> val noether = Person("Emmy", "Noether", 137)
einstein: Person = Person(Emmy,Noether,137)

scala> noether.firstName
res5: String = Emmy

scala> noether.age
res6: Int = 137
```

This data type carries the same information as a tuple `(String, String, Int)`. However, the declaration of a `case class Person` gives the programmer several features that make working with the tuple's data more convenient and less error-prone.

Some (or all) part names may be specified when creating a case class value:

```
scala> val poincaré = Person(firstName = "Henri", lastName = "Poincaré", 165)
poincaré: Person = Person(Henri,Poincaré,165)
```

It is a type error to use wrong types with a case class:

```
scala> val p = Person(140, "Einstein", "Albert")
<console>:13: error: type mismatch;
```

```
 found   : Int(140)
 required: String
       val p = Person(140, "Einstein", "Albert")
                      ^
<console>:13: error: type mismatch;
 found   : String("Albert")
 required: Int
       val p = Person(140, "Einstein", "Albert")
                                       ^
```

Here, the error is due to an incorrect order of parts when creating a case class value. However, parts can be specified in any order when using part names:

```
scala> val p = Person(age = 137, lastName = "Noether", firstName = "Emmy")
p: Person = Person(Emmy,Noether,137)
```

A part of a case class can have the type of another case class, creating a type similar to a nested tuple:

```
case class BagOfSocks(sock: MySock, count: Int)
val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> bag.sock.size
res7: Double = 10.5
```

3.1.2 Case classes with type parameters

Type classes can be defined with type parameters. As an example, consider a generalization of `MySock` where, in addition to the size and color, an "extended sock" holds another value. We could define several specialized case classes,

```
case class MySock_Int(size: Double, color: String, value: Int)
case class MySock_Boolean(size: Double, color: String, value: Boolean)
```

but it is better to define a single parameterized case class

```
case class MySockX[A](size: Double, color: String, value: A)
```

This case class can accommodate every type `A`. We may now create values of `MySockX` containing a `value` of any type,

```
scala> val s = MySockX(10.5, "white", 123)
s: MySockX[Int] = MySockX(10.5,white,123)
```

We see that the type parameter `A` was automatically set to the type `Int`.

Each time we create a value of type `MySockX`, a specific type will have to be used instead of the type parameter `A`. In other words, we can only create values of types `MySockX[Int]`, `MySockX[String]`, etc. If we want to be explicit, we may write

```
scala> val s = MySockX[String](10.5, "white", "last pair")
s: MySockX[String] = MySockX(10.5,white,last pair)
```

However, we can write code working with `MySockX[A]` **parametrically**, that is, keeping the type parameter `A` in the code. For example, a function that checks whether a sock of type `MySockX[A]` fits my foot can be written as

```
def fitsMe[A](sock: MySockX[A]): Boolean = sock.size >= 10.5 && sock.size <= 11.0
```

This function is defined for all types `A` at once, because its code works in the same way regardless of what `A` is. Scala will set the type parameter automatically:

```
scala> fitsMe(MySockX(10.5, "blue", List(1,2,3))) // Parameter A = List[Int]
res0: Boolean = true
```

This code forces the type parameter `A` to be `List[Int]`, and so we may omit the type parameter of `fitsMe`. When types become more complicated, it may be helpful to prevent type errors by specifying the values of some type parameters. For example, here is a type error due to a mismatch between the type parameter `A` used in the "sock" value, which is `List[Int]`, and the type parameter in the function `fitsMe`, specified as `Int`:

```
scala> fitsMe[Int](MySockX(10.5, "blue", List(1,2,3)))
<console>:15: error: type mismatch;
 found   : List[Int]
 required: Int
       fitsMe[Int](MySockX(10.5, "blue", List(1,2,3)))
                                         ^
```

Case classes may have several type parameters, and the types of the parts may use these type parameters. Here is an artificial example of a case class using type parameters in different ways,

```
case class Complicated[A,B,C,D](x: (A, A), y: (B, Int) => A, z: C => C)
```

This case class contains parts of different types that use the type parameters `A`, `B`, `C` in tuples and functions. The type parameter `D` is not used at all; this is allowed.

A type with type parameters, such as `MySockX` or `Complicated`, is called a **type constructor**. A type constructor "constructs" a new type, such as `MySockX[Int]`, from a given type parameter `Int`. Values of type `MySockX` cannot be created without setting the type parameter. So, it is important to distinguish the type constructor, such as `MySockX`, from a type we can use in our code, such as `MySockX[Int]`.

3.1.3 Tuples with one part and with zero parts

Let us compare tuples and case classes more systematically.

Parts of a case class are accessed by name with a dot syntax, for example `sock.color`. Parts of a tuple are accessed with the accessors such as `x._1`. This syntax is the same as that for a case class whose parts have names `_1`, `_2`, etc. So, it appears that tuple parts *do* have names in Scala, although those names are always automatically chosen as `_1`, `_2`, etc. Tuple types are also automatically named in Scala as `Tuple2`, `Tuple3`, etc., and they are parameterized, since each part of the tuple may be of any chosen type. A tuple type expression such as `(Int, String)` is just a special syntax for the parameterized type `Tuple2[Int, String]`. One could define the tuple types as case classes like this,

```
case class Tuple2[A, B](_1: A, _2: B)
case class Tuple3[A, B, C](_1: A, _2: B, _3: C)
// And so on with Tuple4, Tuple5, ...
```

if these types were not already defined in the Scala library.

Proceeding systematically, we ask whether tuple types can have just one part or even no parts. Indeed, Scala defines `Tuple1[A]` as a tuple with a single part. (This type is occasionally used in practice.)

The tuple with zero parts also exists and is called `Unit` (rather than "`Tuple0`"). The syntax for the value of the `Unit` type is the empty tuple, `()`. It is clear that there is *only one* value, `()`, of this type; this explains the name "unit".

At first sight, the `Unit` type may appear to be completely useless: it is a tuple that contains *no data*. It turns out, however, that the `Unit` type is important in functional programming, and it is used as a type *guaranteed* to have only a single distinct value. This chapter will show some examples of using the `Unit` type.

Case classes may have one part or zero parts, similarly to the one-part and zero-part tuples:

```
case class B(z: Int)  // Tuple with one part.
case class C()        // Tuple with no parts.
```

Scala has a special syntax for empty case classes:

```
case object C // Similar to ‘case class C()‘.
```

There are two main differences between `case class C()` and `case object C`:

- A `case object` cannot have type parameters, while we may define, if needed, a `case class C[X, Y, Z]()` with type parameters `X`, `Y`, `Z`.
- A `case object` is allocated in memory only once, while new values of a `case class C()` will be allocated in memory each time `C()` is evaluated.

Other than that, `case class C()` and `case object C` have the same meaning: a named tuple with zero parts, which we may also view as a "named `Unit`" type. In this book, I will not use `case object`s because `case class`es are more general.

Let us summarize the correspondence between tuples and case classes:

Tuples	Case classes
`(123, "xyz"): Tuple2[Int, String]`	`case class A(x: Int, y: String)`
`(123,): Tuple1[Int]`	`case class B(z: Int)`
`(): Unit`	`case class C()`

3.1.4 Pattern matching for case classes

Scala performs pattern matching in two situations:

- destructuring definition: `val` *pattern* `= ...`
- `case` expression: `case` *pattern* `=> ...`

Case classes can be used in both situations. A destructuring definition can be used in a function whose argument is of case class type `BagOfSocks`:

```
case class MySock(size: Double, color: String)
case class BagOfSocks(sock: MySock, count: Int)

def printBag(bag: BagOfSocks): String = {
  val BagOfSocks(MySock(size, color), count) = bag // Destructure the ‘bag‘.
  s"Bag has $count $color socks of size $size"
}

val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> printBag(bag)
res0: String = Bag has 6 white socks of size 10.5
```

An example of using the `case` expression:

```
def fitsMe(bag: BagOfSocks): Boolean = bag match {
  case BagOfSocks(MySock(size, _), _) => size >= 10.5 && size <= 11.0
}
```

In the implementation of this function, we match the `bag` value against the pattern `BagOfSocks(MySock(size, _), _)`. This pattern will always match and will define `size` as a pattern variable of type `Double`.

The syntax for pattern matching expressions with case classes is similar to the syntax for pattern matching of tuples, except for the presence of the *names* of the case classes. For example, removing the case class names from the pattern

```
case BagOfSocks(MySock(size, _), _) => ...
```

we obtain the nested tuple pattern

```
case ((size, _), _) => ...
```

that could be used for values of type `((Double, String), Int)`. We see that case classes behave in many ways exactly as tuple types with names.

Scala's "case classes" got their name from their use in `case` expressions. It is usually more convenient to use `match` / `case` expressions with case classes than to use destructuring.

3.2 Disjunctive types

3.2.1 Motivation and first examples

In many situations, it is useful to have several different shapes of data within the same type. As a first example, suppose we are looking for real roots of a quadratic equation $x^2 + bx + c = 0$. There are three cases: no real roots, one real root, and two real roots. It is convenient to have a type, say `RootsOfQ`, that means "the real roots of a quadratic equation". Inside that type, we distinguish between the three cases, but outside it looks like a single type.

Another example is the binary search algorithm that looks for an integer x in a sorted array. There are two cases: the algorithm either finds the index of x or determines that the array does not contain x. It is convenient if the algorithm could return a single value of a type, say, `SearchResult`, that represents *either* an index at which x is found, *or* the absence of an index.

More generally, we may have computations that *either* return a value *or* generate an error and fail to produce a result. It is then convenient to return a value of type, say, `Result`, that represents either a correct result or an error message.

In certain computer games, one has different types of "rooms", each room having certain properties depending on its type. Some rooms are dangerous because of monsters, other rooms contain useful objects, certain rooms allow you to finish the game, and so on. We want to represent all the different kinds of rooms uniformly, as a type `Room`, so that a value of type `Room` automatically stores the correct properties in each case.

In all these situations, data comes in several mutually exclusive shapes. This data can be represented by a single type if that type is able to describe a mutually exclusive set of cases:

- `RootsOfQ` must be either the empty tuple `()`, or `Double`, or a tuple `(Double, Double)`
- `SearchResult` must be either `Int` or the empty tuple `()`
- `Result` must be either an `Int` value or a `String` message

We see that the empty tuple, also known as the `Unit` type, is natural to use in this representation. It is also helpful to assign names to each of the cases:

- `RootsOfQ` is "no roots" with value `()`, or "one root" with value `Double`, or "two roots" with value `(Double, Double)`
- `SearchResult` is "index" with value `Int`, or "not found" with value `()`

- `Result` is "value" of type `Int` or "error message" of type `String`

Scala's case classes provide exactly what we need here – *named tuples* with zero, one, two and more parts, and so it is natural to use case classes instead of tuples:

- `RootsOfQ` is a value of type `case class NoRoots()`, or a value of type `case class OneRoot(x: Double)`, or of type `case class TwoRoots(x: Double, y: Double)`
- `SearchResult` is a value of type `case class Index(Int)` or a value of type `case class NotFound()`
- `Result` is a value of type `case class Value(x: Int)` or a value of type `case class Error(message: String)`

Our three examples are now described as types that select one case class out of a given set. It remains to see how Scala defines such types. For instance, the definition of `RootsOfQ` needs to indicate that the case classes `NoRoots`, `OneRoot`, and `TwoRoots` are exactly the three alternatives described by the type `RootsOfQ`. The Scala syntax for that definition looks like this:

```
sealed trait RootsOfQ
final case class NoRoots() extends RootsOfQ
final case class OneRoot(x: Double) extends RootsOfQ
final case class TwoRoots(x: Double, y: Double) extends RootsOfQ
```

In the definition of `SearchResult`, we have two cases:

```
sealed trait SearchResult
final case class Index(i: Int) extends SearchResult
final case class NotFound() extends SearchResult
```

The definition of the `Result` type is parameterized, so that we can describe results of any type (while error messages are always of type `String`):

```
sealed trait Result[A]
final case class Value[A](x: A) extends Result[A]
final case class Error[A](message: String) extends Result[A]
```

The "`sealed trait` / `final case class`" syntax defines a type that represents a choice of one case class from a fixed set of case classes. This kind of type is called a **disjunctive type** in this book.

3.2.2 Solved examples: Pattern matching for disjunctive types

Our first examples of disjunctive types are `RootsOfQ`, `SearchResult`, and `Result[A]` defined in the previous section. We will now look at the Scala syntax for creating values of disjunctive types and for using the created values.

Consider the disjunctive type `RootsOfQ` having three case classes (`NoRoots`, `OneRoot`, `TwoRoots`). The only way of creating a value of type `RootsOfQ` is to create a value of one of these case classes. This is done by writing expressions such as `NoRoots()`, `OneRoot(2.0)`, or `TwoRoots(1.0, -1.0)`. Scala will accept these expressions as having the type `RootsOfQ`:

```
scala> val x: RootsOfQ = OneRoot(2.0)
x: RootsOfQ = OneRoot(2.0)
```

Given a value `x:RootsOfQ`, how can we use it, say, as a function argument? The main tool for working with values of disjunctive types is pattern matching with `match` / `case` expressions. In Chapter 2, we used pattern matching to destructure tuples with syntax such as `{ case (x, y) => ... }`. We will now see how to use `match` / `case` expressions with disjunctive types. The main difference is that we may have to write *more than one* `case` pattern in a `match` expression, because we need to match several possible cases of the disjunctive type:

```
def f(r: RootsOfQ): String = r match {
  case NoRoots()       => "no real roots"
  case OneRoot(r)      => s"one real root: $r"
  case TwoRoots(x, y)  => s"real roots: ($x, $y)"
}

scala> f(x)
res0: String = "one real root: 2.0"
```

If we only need to recognize a specific case of a disjunctive type, we can match all other cases with an underscore:

```
scala> x match {
  case OneRoot(r)   => s"one real root: $r"
  case _            => "have something else"
}
res1: String = one real root: 2.0
```

The `match` / `case` expression represents a choice over possible values of a given type. Note the similarity with this code:

```
def f(x: Int): Int = x match {
  case 0    => println(s"error: must be nonzero"); -1
  case 1    => println(s"error: must be greater than 1"); -1
  case _    => x
}
```

The values `0` and `1` are some possible values of type `Int`, just as `OneRoot(1.0)` is a possible value of type `RootsOfQ`. When used with disjunctive types, `match` / `case` expressions will usually contain a complete list of possibilities. If the list of cases is incomplete, the Scala compiler will print a warning:

```
scala> def g(x: RootsOfQ): String = x match {
     |    case OneRoot(r) => s"one real root: $r"
     | }
<console>:14: warning: match may not be exhaustive.
It would fail on the following inputs: NoRoots(), TwoRoots(_, _)
       def g(x: RootsOfQ): String = x match {
                                    ^
```

This code defines a *partial* function `g` that can be applied only to values of the form `OneRoot(...)` and will fail for other values.

Let us look at more examples of using the disjunctive types we just defined.

Example 3.2.2.1 Given a sequence of quadratic equations, compute a sequence containing their real roots as values of type `RootsOfQ`.

Solution Define a case class representing a quadratic equation $x^2 + bx + c = 0$:

```
case class QEqu(b: Double, c: Double)
```

The following function determines how many real roots an equation has:

```
def solve(quadraticEqu: QEqu): RootsOfQ = {
   val QEqu(b, c) = quadraticEqu // Destructure QEqu.
   val d = b * b / 4 - c
   if (d > 0) {
     val s = math.sqrt(d)
     TwoRoots(b / 2 - s, b / 2 + s)
   } else if (d == 0.0) OneRoot(b / 2)
   else NoRoots()
}
```

Test this function:

```
scala> solve(QEqu(1,1))
res1: RootsOfQ = NoRoots()

scala> solve(QEqu(1,-1))
res2: RootsOfQ = TwoRoots(-0.6180339887498949,1.618033988749895)

scala> solve(QEqu(6,9))
res3: RootsOfQ = OneRoot(3.0)
```

We can now implement the required function,

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map(solve)
```

If the function `solve` is not used often, we may want to write it inline:

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map { case QEqu(b, c) =>
  (b * b / 4 - c) match {
    case d if d > 0   =>
      val s = math.sqrt(d)
      TwoRoots(b / 2 - s, b / 2 + s)
    case 0.0          => OneRoot(b / 2)
    case _            => NoRoots()
  }
}
```

This code uses some features of Scala syntax. We can use the partial function `{ case QEqu(b, c) => ... }` directly as the argument of `.map` instead of defining this function separately. This avoids having to destructure `QEqu` at a separate step. The `if` / `else` expression is replaced by an "embedded" `if` within the `case` expression, which is easier to read. Test the final code:

```
scala> findRoots(Seq(QEqu(1,1), QEqu(2,1)))
res4: Seq[RootsOfQ] = List(NoRoots(), OneRoot(1.0))
```

Example 3.2.2.2 Given a sequence of values of type `RootsOfQ`, compute a sequence containing only the single roots. Example test:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = ???

scala> singleRoots(Seq(TwoRoots(-1, 1), OneRoot(3.0), OneRoot(1.0), NoRoots()))
res5: Seq[Double] = List(3.0, 1.0)
```

Solution We apply `.filter` and `.map` to the sequence of roots:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = rs.filter {
  case OneRoot(x) => true
  case _          => false
}.map { case OneRoot(x) => x }
```

In the `.map` operation, we need to cover only the one-root case because the other two possibilities have been "filtered out" by the preceding `.filter` operation.

Example 3.2.2.3 Implement binary search returning a `SearchResult`. We will modify the binary search implementation from Example 2.5.1.4(b) so that it returns a `NotFound` value when appropriate.

Solution The code from Example 2.5.1.4(b) will return *some* index even if the given number is not present in the array:

```
scala> binSearch(Array(1, 3, 5, 7), goal = 5)
res6: Int = 2
```

```
scala> binSearch(Array(1, 3, 5, 7), goal = 4)
res7: Int = 1
```

When the number is not present, the array's element at the computed index will not be equal to `goal`. We should return `NotFound()` in that case. The new code can be written as a `match` / `case` expression for clarity:

```
def safeBinSearch(xs: Seq[Int], goal: Int): SearchResult =
  binSearch(xs, goal) match {
    case n if xs(n) == goal   => Index(n)
    case _                    => NotFound()
  }
```

To test:

```
scala> safeBinSearch(Array(1, 3, 5, 7), 5)
res8: SearchResult = Index(2)

scala> safeBinSearch(Array(1, 3, 5, 7), 4)
res9: SearchResult = NotFound()
```

Example 3.2.2.4 Use the disjunctive type `Result[Int]` to implement "safe integer arithmetic", where a division by zero or a square root of a negative number will give an error message. Define arithmetic operations directly for values of type `Result[Int]`. When errors occur, abandon further computations.

Solution Begin by implementing the square root:

```
def sqrt(r: Result[Int]): Result[Int] = r match {
  case Value(x) if x >= 0  => Value(math.sqrt(x).toInt)
  case Value(x)            => Error(s"error: sqrt($x)")
  case Error(m)            => Error(m) // Keep the error message.
}
```

The square root is computed only if we have the `Value(x)` case, and only if $x \geq 0$. If the argument `r` was already an `Error` case, we keep the error message and perform no further computations.

To implement the addition operation, we need a bit more work:

```
def add(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x + y)
  case (Error(m), _)        => Error(m) // Keep the error message.
  case (_, Error(m))        => Error(m)
}
```

This code illustrates nested patterns that match the tuple `(rx, ry)` against various possibilities. In this way, the code is clearer than code written with nested `if` / `else` expressions.

Implementing the multiplication operation results in almost the same code:

```
def mul(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x * y)
  case (Error(m), _)        => Error(m)
  case (_, Error(m))        => Error(m)
}
```

To avoid repetition, we may define a general function that "lifts" operations on integers to operations on `Result[Int]` types:

```
def do2(rx: Result[Int], ry: Result[Int])(op: (Int, Int) => Int): Result[Int] =
  (rx, ry) match {
    case (Value(x), Value(y)) => Value(op(x, y))
    case (Error(m), _)        => Error(m)
    case (_, Error(m))        => Error(m)
```

```
  }
```

Now we can easily "lift" any binary operation that never generates an error to an operation on `Result[Int]`.

```
def sub(rx: Result[Int], ry: Result[Int]): Result[Int] =
  do2(rx, ry){ (x, y) => x - y }
```

Custom code is still needed for operations that *may* generate errors:

```
def div(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) if y != 0  => Value(x / y)
  case (Value(x), Value(y))            => Error(s"error: $x / $y")
  case (Error(m), _)                   => Error(m)
  case (_, Error(m))                   => Error(m)
}
```

We can now test the new "safe arithmetic" on simple calculations:

```
scala> add(Value(1), Value(2))
res10: Result[Int] = Value(3)

scala> div(add(Value(1), Value(2)), Value(0))
res11: Result[Int] = Error(error: 3 / 0)
```

We see that indeed all further computations are abandoned once an error occurs. An error message shows only the immediate calculation that generated the error. For instance, the error message for $20 + 1/0$ never mentions 20:

```
scala> add(Value(20), div(Value(1), Value(0)))
res12: Result[Int] = Error(error: 1 / 0)

scala> add(sqrt(Value(-1)), Value(10))
res13: Result[Int] = Error(error: sqrt(-1))
```

3.2.3 Standard disjunctive types: `Option`, `Either`, `Try`

The Scala library defines the disjunctive types `Option`, `Either`, and `Try` because they are used often. We now look at each of them in turn.

`Option` The `Option` type is a disjunctive type with two cases: the empty tuple and a one-element tuple. The names of the two case classes are `None` and `Some`. If the `Option` type were not already defined in the standard library, one could define it with the code

```
sealed trait Option[T]
final case object None extends Option[Nothing]
final case class Some[T](t: T) extends Option[T]
```

This code is similar to the type `SearchResult` defined in Section 3.2.1, except that `Option` has a type parameter instead of a fixed type `Int`. Another difference is the use of a `case object` for the empty case instead of an empty case class, such as `None()`. Since Scala's `case object`s cannot have type parameters, the type parameter in the definition of `None` must be set to the special type `Nothing`, which is a type with *no* values (also known as the **void** type).

An alternative (implemented in libraries such as `scalaz`) is to define

```
final case class None[T]() extends Option[T]
```

and write the empty option value as `None()`. In that implementation, the empty option also has a type parameter.

Several consequences follow from the Scala library's decision to define `None` without a type parameter. One consequence is that the single value `None` can be reused as a value of type `Option[A]` for any type `A`:

```
scala> val y: Option[Int] = None
y: Option[Int] = None

scala> val z: Option[String] = None
z: Option[String] = None
```

Typically, `Option` is used in situations where a value may be either present or missing, especially when a missing value is *not an error*. The missing-value case is represented by `None`, while `Some(x)` means that a value `x` is present.

Example 3.2.3.1 Suppose that information about subscribers to a certain online service must contain a name and an email address, but a telephone number is optional. To represent this information, we may define a case class like this,

```
case class Subscriber(name: String, email: String, phone: Option[Long])
```

What if we represent the missing telephone number by a special value such as `-1` and use the simpler type `Long` instead of `Option[Long]`? The disadvantage is that we would need to *remember* to check for the special value `-1` in all functions that take the telephone number as an argument. Looking at a function such as `sendSMS(phone: Long)` at a different place in the code, a programmer might forget that the telephone number is actually optional. In contrast, the type signature `sendSMS(phone: Option[Long])` unambiguously indicates that the telephone number might be missing and helps the programmer to remember that case.

Pattern-matching code involving `Option` needs two cases:

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone match {
  case None               => None   // Do nothing.
  case Some(number)       => Some(digitsOf(number))
}
```

Here we used the function `digitsOf` defined in Section 2.3.

At the two sides of `case None => None`, the value `None` has different types, namely `Option[Long]` and `Option[Seq[Long]]`. Since these types are declared in the type signature of the function `getDigits`, the Scala compiler is able to figure out the types of all expressions in the `match` / `case` construction. So, pattern-matching code can be written without explicit type annotations such as `(None: Option[Long])`.

If we now need to compute the number of digits, we can write

```
def numberOfDigits(phone: Option[Long]): Option[Long] = getDigits(phone) match {
  case None               => None   // Do nothing.
  case Some(digits)       => Some(digits.length)
}
```

These examples perform a computation when an `Option` value is non-empty, and leave it empty otherwise. To avoid repeating this kind of code, we can implement this design pattern as a function that takes the computation as a parameter:

```
def doComputation(x: Option[Long], f: Long => Long): Option[Long] = x match {
  case None               => None   // Do nothing.
  case Some(i)            => Some(f(i))
}
```

It is then natural to generalize this function to arbitrary types using type parameters instead of a fixed type `Long`. The resulting function is usually called `fmap`:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None               => None   // Do nothing.
  case Some(a)            => Some(f(a))
}

scala> fmap(digitsOf)(Some(4096))
res0: Option[Seq[Long]] = Some(List(4, 0, 9, 6))

scala> fmap(digitsOf)(None)
res1: Option[Seq[Long]] = None
```

One can say that the `fmap` operation **lifts** a given function of type `A => B` to the type `Option[A] => Option[B]`.

The Scala library implements an equivalent function as a method on the `Option` class, with the syntax `x.map(f)` rather than `fmap(f)(x)`. We can concisely rewrite the previous code using the standard library methods as

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone.map(digitsOf)
def numberOfDigits(phone: Option[Long]): Option[Long] = phone.map(digitsOf).map(_.length)
```

We see that the `.map` operation for the `Option` type is analogous to the `.map` operation for sequences.

The similarity between `Option[A]` and `Seq[A]` is made clearer if we view `Option[A]` as a special kind of "sequence" whose length is restricted to be either 0 or 1. So, `Option[A]` can have all the operations of `Seq[A]`, except the operations such as `.concat` that may increase the length of the sequence. The standard operations defined on `Option` include `.map`, `.filter`, `.forall`, `.exists`, `.flatMap`, and `.foldLeft`.

Example 3.2.3.2 Given a phone number as `Option[Long]`, extract the country code if it is present. (Assume that the country code is any digits in front of the 10-digit number; for the phone number 18004151212, the country code is 1.) The result must be again of type `Option[Long]`.

Solution If the phone number is a positive integer n, we may compute the country code simply as `n / 10000000000L`. However, if the result of that division is zero, we should return an empty `Option` (i.e. the value `None`) rather than `0`. To implement this logic, we may begin by writing this code,

```
def countryCode(phone: Option[Long]): Option[Long] = phone match {
  case None       => None
  case Some(n)    =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}
```

We may notice that we have reimplemented the design pattern similar to `.map` in this code, namely "if `None`, return `None`, else do a computation". So we may try to rewrite the code as

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Type error: the result is Option[Option[Long]], not Option[Long].
```

This code does not compile: we are returning an `Option[Long]` within a function lifted via `.map`, so the resulting type is `Option[Option[Long]]`. We may use `.flatten` to convert `Option[Option[Long]]` to the required type `Option[Long]`,

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}.flatten // Types are correct now.
```

Since the `.flatten` follows a `.map`, we can rewrite the code using `.flatMap`:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.flatMap { n =>
```

```
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Types are correct now.
```

Another way of implementing this example is to notice the design pattern "if condition does not hold, return `None`, otherwise keep the value". For an `Option` type, this is equivalent to the `.filter` operation (recall that `.filter` returns an empty sequence if the predicate never holds). So the final code is

```
def countryCode(phone: Option[Long]): Option[Long] =
  phone.map(_ / 10000000000L).filter(_ != 0L)
```

Test it:

```
scala> countryCode(Some(18004151212L))
res0: Option[Long] = Some(1)

scala> countryCode(Some(8004151212L))
res1: Option[Long] = None
```

Example 3.2.3.3 Add a new requirement to the phone number example: if the country code is not present, we should return the default country code 1. This is an often used design pattern: "if empty, substitute a default value". The Scala library has the method `.getOrElse` for this purpose:

```
scala> Some(100).getOrElse(1)
res2: Int = 100

scala> None.getOrElse(1)
res3: Int = 1
```

So we can implement the new requirement as

```
scala> countryCode(Some(8004151212L)).getOrElse(1L)
res4: Long = 1
```

Using `Option` with collections Many Scala library methods return an `Option` as a result. The main examples are `.find`, `.headOption`, and `.lift` for sequences, and `.get` for dictionaries.

The `.find` method returns the first element satisfying a predicate:

```
scala> (1 to 10).find(_ > 5)
res0: Option[Int] = Some(6)

scala> (1 to 10).find(_ > 10) // No element is > 10.
res1: Option[Int] = None
```

The `.lift` method returns the element of a sequence at a given index:

```
scala> (10 to 100).lift(0)
res2: Option[Int] = Some(10)

scala> (10 to 100).lift(1000) // No element at index 1000.
res3: Option[Int] = None
```

The `.headOption` method returns the first element of a sequence, unless the sequence is empty. This is equivalent to `.lift(0)`:

```
scala> Seq(1,2,3).headOption
res4: Option[Int] = Some(1)

scala> Seq(1,2,3).filter(_ > 10).headOption
```

```
res5: Option[Int] = None
```

Applying `.find(p)` computes the same result as `.filter(p).headOption`, but `.find(p)` may be more efficient.

The `.get` method for a dictionary returns the value if it exists for a given key, and returns `None` if the key is not in the dictionary:

```
scala> Map(10 -> "a", 20 -> "b").get(10)
res6: Option[String] = Some(a)

scala> Map(10 -> "a", 20 -> "b").get(30)
res7: Option[String] = None
```

The `.get` method provides safe by-key access to dictionaries, unlike the direct access method that may fail:

```
scala> Map(10 -> "a", 20 -> "b")(10)
res8: String = a

scala> Map(10 -> "a", 20 -> "b")(30)
java.util.NoSuchElementException: key not found: 30
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  ... 32 elided
```

Similarly, `.lift` provides safe by-index access to collections, unlike the direct access that may fail:

```
scala> Seq(10,20,30)(0)
res9: Int = 10

scala> Seq(10,20,30)(5)
java.lang.IndexOutOfBoundsException: 5
  at scala.collection.LinearSeqOptimized$class.apply(LinearSeqOptimized.scala:65)
  at scala.collection.immutable.List.apply(List.scala:84)
  ... 32 elided
```

`Either` The standard disjunctive type `Either[A, B]` has two type parameters and is often used for computations that report errors. By convention, the *first* type (`A`) is the type of error, and the *second* type (`B`) is the type of the (non-error) result. The names of the two cases are `Left` and `Right`. A possible definition of `Either` may be written as

```
sealed trait Either[A, B]
final case class  Left[A, B](value: A) extends Either[A, B]
final case class Right[A, B](value: B) extends Either[A, B]
```

By convention, a value `Left(x)` is used to represent an error, and a value `Right(y)` is used to represent a valid result.

As an example, the following function substitutes a default value and logs the error information:

```
def logError(x: Either[String, Int], default: Int): Int = x match {
  case Left(error)  => println(s"Got error: $error"); default
  case Right(res)   => res
}
```

To test:

```
scala> logError(Right(123), -1)
res1: Int = 123

scala> logError(Left("bad result"), -1)
Got error: bad result
```

```
res2: Int = -1
```

Why use `Either` instead of `Option` for computations that may fail? A failing computation such as `1/0` could return `None` to indicate that the result is not available. However, the result is not an optional value that may be missing; usually the result is required, and if it is not available, we need to know exactly *which* error prevented the result from being available. The `Either` type provides the possibility to store information about the error, which `Option` does not provide.

The `Either` type generalizes the type `Result` defined in Section 3.2.1 with an arbitrary error type instead of `String`. We have seen its usage in Example 3.2.2.4, where the design pattern was "if value is present, do a computation, otherwise keep the error". This design pattern is implemented by the `.map` method on `Either`:

```
scala> Right(1).map(_ + 1)
res0: Either[Nothing, Int] = Right(2)

scala> Left[String, Int]("error").map(_ + 1)
res1: Either[String, Int] = Left("error")
```

The type `Nothing` was filled in by the Scala compiler because we did not specify the full type of `Either` in the first line.

The methods `.filter`, `.flatMap`, `.fold`, and `.getOrElse` are also defined for the `Either` class, with the same convention that a `Left` value represents an error.[1]

Exceptions and the `Try` type When computations fail for any reason, Scala generates an **exception** instead of returning a value. An exception means that the evaluation of the expression was stopped without returning a result.

An exception is generated when the available memory is too small to store the resulting data (as we saw in Section 2.6.3), or if a stack overflow occurs during the computation (as we saw in Section 2.2.3). Exceptions may also occur due to programmer's error: when a pattern matching operation fails, when a requested key does not exist in a dictionary, or when the `.head` operation is applied to an empty list.

Motivated by these examples, we may distinguish "planned" and "unplanned" exceptions.

A **planned** exception is generated by programmer's code via the `throw` syntax:

```
scala> throw new Exception("this is a test... this is only a test")
java.lang.Exception: this is a test... this is only a test
  ... 42 elided
```

The Scala library contains a `throw` operation in various places, such as in the code for applying the `.head` method to an empty sequence, as well as in other situations where exceptions are generated due to programmer's errors. These exceptions are generated deliberately and in well-defined situations. Although these exceptions indicate errors, these errors are anticipated in advance and so may be handled by the programmer.

For example, many Java libraries will generate exceptions when function arguments have unexpected values, when a network operation takes too long or fails to make a network connection, when a file is not found or cannot be read, and in many other situations. All these exceptions are "planned" because they are generated explicitly by library code such as `throw new FileNotFoundException(...)`. The programmer's code is expected to catch these exceptions, to handle the problem, and to continue the evaluation of the program.

An **unplanned** exception is generated by the Java runtime system when critical errors occur, such as an out-of-memory error. It is rare that a programmer writes `val y = f(x)` while *expecting* that an out-of-memory exception will sometimes occur at that point. An unplanned exception indicates a serious and unforeseen problem with memory or another critically important resource, such as

[1] These methods are available in Scala 2.12 or a later version.

the operating system's threads or file handles. Such problems usually cannot be fixed and will prevent the program from running any further. It is reasonable that the program evaluation should immediately stop (or "crash" as programmers say) after such an error.

The use of planned exceptions assumes that the programmer will write code to handle each exception. This assumption makes it significantly harder to write programs correctly. It is hard to figure out and to keep in mind all the possible exceptions that a given library function may `throw` in its code (or in the code of all other libraries on which it depends). Instead of using exceptions for indicating errors, Scala programmers can write functions that return a disjunctive type such as `Either`, describing both the correct result and a possible error condition. Users of these functions will need to do pattern matching on the results, which indicates unambiguously both the possible presence of errors and the kinds of errors that need to be handled.

However, the programmer will often need to use Java libraries (or Scala libraries) that `throw` exceptions. To help write code for these situations, the Scala library contains a helper function called `Try()` and a disjunctive type also called `Try`. The type `Try[A]` can be seen as similar to `Either[Throwable, A]`, where `Throwable` is the general type of all exceptions (i.e. values to which a `throw` operation can be applied). The two parts of the disjunctive type `Try[A]` are called `Failure` and `Success[A]` (instead of `Left[Throwable]` and `Right[A]` in the `Either` type). The function `Try(expr)` will catch all exceptions `thrown` while the expression `expr` is evaluated. If the evaluation of `expr` succeeds and returns a value `x:A`, the value of `Try(expr)` will be `Success(x)`. Otherwise it will be `Failure(t)`, where `t:Throwable` is the value associated with the generated exception. Here is an example of using `Try`:

```
import scala.util.{Try,Success,Failure}

scala> Try(1 / 0)
res0: Try[Int] = Failure(java.lang.ArithmeticException: / by zero)

scala> Try(1 + 1)
res1: Try[Int] = Success(2)
```

Unlike computing `1/0` without an enclosing `Try()`, the computation `Try(1/0)` does not generate any exceptions and will not crash the program. Any computation that may `throw` an exception can be enclosed in a `Try()`, and the exception will be caught and encapsulated within the disjunctive type as a `Failure(...)` value.

The methods `.map`, `.filter`, `.flatMap`, `.foldLeft` are defined for the `Try` class similarly to the `Either` type. One additional feature of `Try` is to catch exceptions generated by the function arguments of `.map`, `.filter`, etc.:

```
scala> val x = Try(1)
x: scala.util.Try[Int] = Success(1)

scala> val y: Try[Int] = x.map(y => throw new Exception("test1"))
y: scala.util.Try[Int] = Failure(java.lang.Exception: test1)

scala> val z = x.filter(y => throw new Exception("test1"))
z: scala.util.Try[Int] = Failure(java.lang.Exception: test1)
```

In this example, the values `y` and `z` were computed *successfully* even though exceptions were thrown while the function arguments of `.map` and `.filter` were evaluated. Other code can use pattern matching on the values `y` and `z` and determine which exceptions occurred. However, it is important that these exceptions were caught and the other code is *able* to run.

Another useful method is `.toOption`; it will discard the error information:

```
scala> Try(1 / 0).toOption
res2: Option[Int] = None

scala> Try(1 + 1).toOption
res3: Option[Int] = Some(2)
```

Instead of exceptions, programmers can use *values* of type `Try` or of other disjunctive types in order to represent all anticipated failures or errors. Representing all errors by ordinary Scala values gives us assurance that the program will not crash because of an exception that we did not know about or forgot to handle.

3.3 Lists and trees: recursive disjunctive types

Consider this code:

```
sealed trait NInt
final case class One(x: Int) extends NInt
final case class Two(n: NInt) extends NInt
```

We are defining a new disjunctive type `NInt`, but the case class `Two` uses the type `NInt` as if it were already defined. Scala allows us to write such definitions.

A type whose definition uses that same type is called a **recursive type**. So, `NInt` is a recursive disjunctive type.

We might imagine a disjunctive type with many case classes whose parts are recursively using the same type in complicated ways. What would this data type be useful for, and what kind of data does it represent? In general, this question is not easy to answer. For instance, the simple definition

```
final case class Bad(x: Bad)
```

is useless: to create a value of type `Bad` we already need to have a value of type `Bad`. This is an example of an infinite type recursion. We will never be able to create any values of type `Bad`, which means that the type `Bad` is effectively **void** (has no values, like the the special type `Nothing`).

Chapter **??** studies recursive types in more detail. For now, we will look at the main examples of recursive disjunctive types that are *known* to be useful. These examples are lists and trees.

3.3.1 Lists

A list of values of type `A` is either empty, or one value of type `A`, or two values of type `A`, etc. We can visualize the type `List[A]` as a disjunctive type defined by

```
sealed trait List[A]
final case class List0[A]() extends List[A]
final case class List1[A](x: A) extends List[A]
final case class List2[A](x1: A, x2: A) extends List[A]
??? // Need an infinitely long definition.
```

However, this definition is not practical – we cannot define a separate case class for a list of *each* possible length. Instead, we define the type `List[A]` via mathematical induction on the length of the list:

- Base case: empty list, `case class List0[A]()`.
- Inductive step: given a list of a previously defined length, say $\texttt{List}_{n-1}$, define a new case class $\texttt{List}_n$ describing a list with one more element of type `A`. So we could define $\texttt{List}_n = (\texttt{List}_{n-1}, \texttt{A})$.

Let us try to write this inductive definition as code:

```
sealed trait ListI[A] // Inductive definition of a list.
final case class List0[A]() extends ListI[A]
final case class List1[A](prev: List0[A], x: A) extends ListI[A]
final case class List2[A](prev: List1[A], x: A) extends ListI[A]
```

```
??? // Still need an infinitely long definition.
```

To avoid writing an infinitely long type definition, we need to use a trick. Notice that all definitions of `List1`, `List2`, etc., have a similar form (while `List0` is not similar). We can replace all the definitions `List1`, `List2`, etc., by a single definition if we use the type `ListI[A]` recursively inside the case class:

```
sealed trait ListI[A] // Inductive definition of a list.
final case class List0[A]() extends ListI[A]
final case class ListN[A](prev: ListI[A], x: A) extends ListI[A]
```

The type definition has become recursive. For this trick to work, it is important to use `ListI[A]` and not `ListN[A]` inside the definition `ListN[A]`; or else we would have created an infinite type recursion similar to `case class Bad` shown above.

Since we obtained the type definition of `ListI` via a trick, let us verify that the code actually defines the disjunctive type we wanted.

To create a value of type `ListI[A]`, we must use one of the two available case classes. Using the first case class, we may create a value `List0()`. Since this empty case class does not contain any values of type `A`, it effectively represents an empty list (the base case of the induction). Using the second case class, we may create a value `ListN(prev, x)` where `x` is of type `A` and `prev` is some previously constructed value of type `ListI[A]`. This represents the induction step, because the case class `ListN` is a named tuple containing `ListI[A]` and `A`. Now, the same consideration recursively applies to constructing the value `prev`, which must be either an empty list or a pair containing another list and an element of type `A`. The assumption that the value `prev:ListI[A]` is already constructed is equivalent to the inductive assumption that we already have a list of a previously defined length. So, we have verified that `ListI[A]` implements the inductive definition shown above.

Examples of values of type `ListI` are the empty list `List0()`, a one-element list `ListN(List0(), x)`, and a two-element list `ListN(ListN(List0(), x), y)`.

To illustrate writing pattern-matching code using this type, let us implement the method `headOption`:

```
@tailrec def headOption[A]: ListI[A] => Option[A] = {
  case List0()              => None
  case ListN(List0(), x)    => Some(x)
  case ListN(prev, _)       => headOption(prev)
}
```

The Scala library already defines the type `List[A]`, but its case classes are named differently, and the second case class uses the name `::` with an infix syntax and places the value of type `A` *before* the previously constructed list,

```
sealed trait List[A]
final case object Nil extends List[Nothing]
final case class ::[A](head: A, tail: List[A]) extends List[A]
```

Because "operator-like" case class names, such as `::`, support the infix syntax, we may write `head :: tail` instead of `::(head, tail)`. Pattern matching with the standard `List` class looks like this:

```
def headOption[A]: List[A] => Option[A] = {
  case Nil           => None
  case head :: tail  => Some(head)
}
```

Examples of values created using Scala's standard `List` type are the empty list `Nil`, a one-element list `x :: Nil`, and a two-element list `x :: y :: Nil`. We see that list values are easier to read in the standard syntax. The same syntax such as `x :: y :: Nil` is used both for creating values of type `List` and for pattern-matching on such values.

The Scala library also defines the helper function `List()`, so that `List()` is the same as `Nil` and

`List(1, 2, 3)` is the same as `1 :: 2 :: 3 :: Nil`.

3.3.2 Tail recursion with `List`

Because the `List` type is defined by induction, it is straightforward to implement iterative computations with the `List` type using recursion.

A first example is the `map` function. We use reasoning by induction in order to figure out the implementation of `map`. The required type signature is

```
def map[A, B](xs: List[A])(f: A => B): List[B] = ???
```

The base case is an empty list, and we return again an empty list:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil => Nil
  ...
```

In the induction step, we have a pair `(head, tail)` in the case class `::`, where `head:A` and `tail:List[A]`. The pair can be pattern-matched with the syntax `head :: tail`. The `map` function should apply the argument `f` to the head value, which will give the first element of the resulting list. The remaining elements are computed by the induction assumption, i.e. by a recursive call to `map`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil           => Nil
  case head :: tail  => f(head) :: map(tail)(f) // Not tail-recursive.
```

While this implementation is straightforward and concise, it is not tail-recursive. This will be a problem for large enough lists.

Instead of implementing the often-used methods such as `.map` or `.filter` one by one, let us implement `foldLeft` because most of the other methods can be expressed via `foldLeft`.

The required type signature is

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = ???
```

Reasoning by induction, we start with the base case, where we have an empty list, and the only possibility is to return the value `init`.

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = xs match {
    case Nil            => init
    ...
```

The induction step for `foldLeft` says that, given the values `head:A` and `tail:List[A]`, we need to apply the updater function to the previous accumulator value. That value is `init`. So we apply `foldLeft` recursively to the tail of the list once we have the updated accumulator value:

```
@tailrec def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R =
  xs match {
    case Nil            => init
    case head :: tail   =>
      val newInit = f(init, head) // Update the accumulator.
      foldLeft(tail)(newInit)(f)  // Recursive call to ‘foldLeft‘.
  }
```

This implementation is tail-recursive because the recursive call to `foldLeft` is the last expression returned in its `case` branch.

Another example is a function for reversing a list. The Scala library defines the `.reverse` method for this task, but we will show an implementation using `foldLeft`. The updater function *prepends* an element to a previous list:

```
def reverse[A](xs: List[A]): List[A] =
  xs.foldLeft(Nil: List[A])((prev, x) => x :: prev)

scala> reverse(List(1, 2, 3))
res0: List[Int] = List(3, 2, 1)
```

Without the explicit type annotation `Nil:List[A]`, the Scala compiler will decide that `Nil` has type `List[Nothing]`, and the types will not match later in the code.

The `reverse` function can be used to obtain a tail-recursive implementation of `map` for `List`. The idea is to first use `foldLeft` to accumulate transformed elements:

```
scala> Seq(1, 2, 3).foldLeft(Nil:List[Int])((prev, x) => x*x :: prev)
res0: List[Int] = List(9, 4, 1)
```

The result is a reversed `.map(x => x*x)`, so we reverse that:

```
def map[A, B](xs: List[A])(f: A => B): List[B] =
  xs.foldLeft(Nil: List[B])((prev, x) => f(x) :: prev).reverse

scala> map(List(1, 2, 3))(x => x*x)
res2: List[Int] = List(1, 4, 9)
```

This achieves stack safety at the cost of traversing the list twice. (This implementation is shown only as an example. The Scala library implements `.map` for `List` using low-level tricks in order to achieve better performance.)

Example 3.3.2.1 A definition of the **non-empty list** is similar to `List` except that the empty-list case is replaced by the 1-element case:

```
sealed trait NEL[A]
final case class Last[A](head: A) extends NEL[A]
final case class More[A](head: A, tail: NEL[A]) extends NEL[A]
```

Values of a non-empty list look like this:

```
scala> val xs: NEL[Int] = More(1, More(2, Last(3))) // [1, 2, 3]
xs: NEL[Int] = More(1,More(2,Last(3)))

scala> val ys: NEL[String] = Last("abc") // One element.
ys: NEL[String] = Last(abc)
```

To create non-empty lists more easily, we implement a conversion function from an ordinary list. Since the conversion function must guarantee that the result is a non-empty list, we give it two arguments:

```
def toNEL[A](x: A, rest: List[A]): NEL[A] = rest match {
  case Nil        => Last(x)
  case y :: tail  => More(x, toNEL(y, tail))
} // Not tail-recursive: `toNEL()` is used inside `More(...)`.
```

To test:

```
scala> toNEL(1, List()) // Result = [1].
res0: NEL[Int] = Last(1)

scala> toNEL(1, List(2, 3)) // Result = [1, 2, 3].
res1: NEL[Int] = More(1,More(2,Last(3)))
```

The `head` method is safe for non-empty lists, unlike `.head` for ordinary `List`s:

```
def head[A]: NEL[A] => A = {
```

```
    case Last(x)        => x
    case More(x, _)     => x
}
```

We can also implement a tail-recursive `foldLeft` function for non-empty lists:

```
@tailrec def foldLeft[A, R](n: NEL[A])(init: R)(f: (R, A) => R): R = n match {
    case Last(x)        => f(init, x)
    case More(x, tail)  => foldLeft(tail)(f(init, x))(f)
  }

scala> foldLeft(More(1, More(2, Last(3))))(0)(_ + _)
res2: Int = 6
```

Example 3.3.2.2 Use `foldLeft` to implement a `reverse` function for the type `NEL`. The required type signature and a sample test:

```
def reverse[A]: NEL[A] => NEL[A] = ???

scala> reverse(toNEL(1, List(2, 3))) // Result must be [3, 2, 1].
res3: NEL[Int] = More(3, More(2, Last(1)))
```

Solution We will use `foldLeft` to build up the reversed list as the accumulator value. It remains to choose the initial value of the accumulator and the updater function. We have already seen the code for reversing the ordinary list using `foldLeft` operation (Section 3.3.2),

```
def reverse[A](xs: List[A]): List[A] =
  xs.foldLeft(Nil: List[A])((prev,x) => x :: prev)
```

However, we cannot reuse the same code for non-empty lists by writing `More(x, prev)` instead of `x :: prev`, because the `foldLeft` operation works with non-empty lists differently. Since a list element is always available, the updater function is always applied to the initial value, and the code works incorrectly:

```
def reverse[A](xs: NEL[A]): NEL[A] =
  foldLeft(xs)(Last(head(xs)):NEL[A])((prev,x) => More(x, prev))

scala> reverse1(toNEL(1,List(2,3))) // Result = [3, 2, 1, 1].
res4: NEL[Int] = More(3,More(2,More(1,Last(1))))
```

The last element, `1`, should not have been repeated. It was repeated because the initial accumulator value already contained the head element `1` of the original list. However, we cannot set the initial accumulator value to an empty list, since a value of type `NEL[A]` must be non-empty. It seems that we need to handle the case of a one-element list separately. So we begin by matching on the argument of `reverse`, and apply `foldLeft` only when the list is longer than 1 element:

```
def reverse[A]: NEL[A] => NEL[A] = {
    case Last(x)         => Last(x)  // Trivial reverse.
    case More(x, tail)   => // Use foldLeft on ‘tail‘.
      foldLeft(tail)(Last(x):NEL[A])((prev,x) => More(x, prev))
  }

scala> reverse(toNEL(1, List(2, 3))) // Result = [3, 2, 1].
res5: NEL[Int] = More(3, More(2, Last(1)))
```

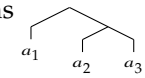
Exercise 3.3.2.3 Implement a function `toList` that converts a non-empty list into an ordinary Scala `List`. The required type signature and a sample test:

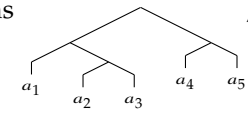
```
def toList[A](nel: NEL[A]): List[A] = ???
```

```
scala> toList(More(1, More(2, Last(3)))) // This is [1, 2, 3].
res4: List[Int] = List(1, 2, 3)
```

3.3.3 Binary trees

We will consider four kinds of trees defined as recursive disjunctive types: binary trees, rose trees, homogeneous trees, and abstract syntax trees.

Examples of a **binary tree** with leaves of type `A` can be drawn as 

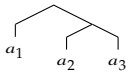
or as 

,

where a_i are some values of type `A`.

An inductive definition says that a binary tree is either a leaf with a value of type `A` or a branch containing *two* previously defined binary trees. Translating this definition into code, we get

```
sealed trait Tree2[A]
final case class Leaf[A](a: A) extends Tree2[A]
final case class Branch[A](x: Tree2[A], y: Tree2[A]) extends Tree2[A]
```

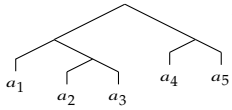
The tree 

is created by the code

```
Branch(Leaf("a1"), Branch(Leaf("a2"), Leaf("a3")))
```

while the code

```
Branch(Branch(Leaf("a1"),Branch(Leaf("a2"), Leaf("a3"))), Branch(Leaf("a4"), Leaf("a5")))
```

creates the tree 

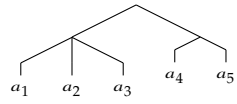
.

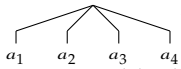
Recursive functions on trees are translated into concise code. For instance, the function `foldLeft` for trees of type `Tree2` is defined by

```
def foldLeft[A, R](t: Tree2[A])(init: R)(f: (R, A) => R) = t match {
  case Leaf(a)          => f(init, a)
  case Branch(t1, t2)   =>
    val r1 = foldLeft(t1)(init)(f) // Fold the left branch.
    foldLeft(t2)(r1)(f) // Starting from ‘r1‘, fold the right branch.
}
```

Note that this function cannot be made tail-recursive using the accumulator trick, because `foldLeft` needs to call itself twice in the `Branch` case.

3.3.4 Rose trees

A **rose tree** is similar to the binary tree except the branches contain a non-empty list of trees. Because of that, a rose tree can fork into arbitrarily many branches at each node, rather than always into two branches as the binary tree does. Example shapes for a rose tree are 

and 

.

A possible definition of a data type for the rose tree is

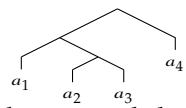
```
sealed trait TreeN[A]
final case class Leaf[A](a: A) extends TreeN[A]
final case class Branch[A](ts: NEL[TreeN[A]]) extends TreeN[A]
```

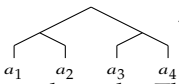
Exercise 3.3.4.1 Define the function `foldLeft` for a rose tree, using `foldLeft` for the type `NEL`. Type signature and a test:

```
def foldLeft[A, R](t: TreeN[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(More(Leaf(1), More(Leaf(2), Last(Leaf(3))))))(0)(_ + _)
res0: Int = 6
```

3.3.5 Regular-shaped trees

Binary trees and rose trees may choose to branch or not to branch at any given node, resulting in structures that may have different branching depths at different nodes, such as 

. A **regular-shaped tree** always branches in the same way at every node until a chosen total depth, e.g. 

, where all nodes at depth 0 and 1 always branch into two, while nodes at depth 2 never branch. The branching number is fixed for a given type of a regular-shaped tree; in this example, the branching number is 2, so it is a regular-shaped *binary* tree.

How can we define a data type representing a regular-shaped binary tree? We need a tree that is either a single value, or a pair of values, or a pair of pairs, etc. Begin with the non-recursive (but, of course, impractical) definition

```
sealed trait RTree[A]
final case class Leaf[A](x: A) extends RTree[A]
final case class Branch1[A](xs: (A, A)) extends RTree[A]
final case class Branch2[A](xs: ((A, A),(A, A))) extends RTree[A]
??? // Need an infinitely long definition.
```

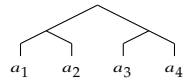
The case `Branch1` describes a regular-shaped tree with total depth 1, the case `Branch2` has total depth 2, and so on. Now, we cannot rewrite this definition as a recursive type because the case classes do not have the same structure. The non-trivial trick is to notice that each $\texttt{Branch}_n$ case class uses the previous case class's data structure *with the type parameter* set to `(A, A)` instead of `A`. So we can rewrite this definition as

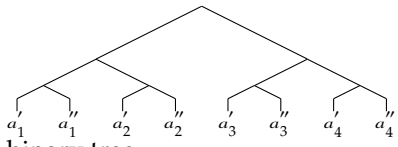
```
sealed trait RTree[A]
final case class Leaf[A](x: A) extends RTree[A]
final case class Branch1[A](xs: Leaf[(A, A)]) extends RTree[A]
final case class Branch2[A](xs: Branch1[(A, A)]) extends RTree[A]
??? // Need an infinitely long definition.
```

We can now apply the type recursion trick: replace the type $\texttt{Branch}_{n-1}\texttt{[(A, A)]}$ in the definition of $\texttt{Branch}_n$ by the type `RTree[(A, A)]` . This gives the type definition for a regular-shaped binary tree:

```
sealed trait RTree[A]
final case class Leaf[A](x: A) extends RTree[A]
final case class Branch[A](xs: RTree[(A, A)]) extends RTree[A]
```

Since we used some tricks to figure out the definition of `RTree[A]`, let us verify that this definition actually describes the recursive disjunctive type we wanted. The only way to create a struc-

ture of type `RTree[A]` is either to have a `Leaf[A]` or a `Branch[A]`. A value of type `Leaf[A]` is a correct regularly-shaped tree; it remains to consider the case of `Branch[A]`. To create a `Branch[A]` requires a previously created `RTree` with values of type `(A, A)` instead of `A`. By the inductive assumption, the previously created `RTree[A]` would have the correct shape. Now, it is clear that if we replace the type parameter `A` by the pair `(A, A)`, a regular-shaped tree such as 

remains regular-shaped but becomes one level deeper, which can be drawn (replacing each a_i by a pair a_i', a_i'') as 

. We see that `RTree[A]` is the correct definition of a regular-shaped binary tree.

Example 3.3.5.1 Define a (non-tail-recursive) `map` function for a regular-shaped binary tree. The required type signature and a test:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = ???

scala> map(Branch(Branch(Leaf(((1,2),(3,4))))))(_ * 10)
res0: RTree[Int] = Branch(Branch(Leaf(((10,20),(30,40)))))
```

Solution Begin by pattern-matching on the tree:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => ???
  case Branch(xs)   => ???
}
```

In the base case, we have no choice but to return `Leaf(f(x))`.

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => ???
}
```

In the inductive step, we are given a previous tree value `xs:RTree[(A, A)]`. It is clear that we need to apply `map` recursively to `xs`. Let us try:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs)(f)) // Type error!
}
```

Here, `map(xs)(f)` does not compile because the type of the function `f` is incorrect. Since `xs` has type `RTree[(A, A)]`, the recursive call `map(xs)(f)` requires `f` to be of type `((A, A)) => (B, B)` instead of `f: A => B`. So, we need to provide a function of the correct type instead of `f`. A function of type `((A, A)) => (B, B)` will be obtained out of `f: A => B` if we apply `f` to each part of the tuple `(A, A)`; the code for that function is `{case (x, y) => (f(x), f(y))}`. Therefore, we can implement `map` as

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)      => Leaf(f(x))
  case Branch(xs)   => Branch(map(xs){ case (x, y) => (f(x), f(y)) })
}
```

Exercise 3.3.5.2 Using tail recursion, compute the depth of a regular-shaped binary tree of type `RTree`. (An `RTree` of depth n has 2^n leaf values.) The required type signature and a test:

```
@tailrec def depth[A](t: RTree[A]): Int = ???

scala> depth(Branch(Branch(Leaf((("a","b"),("c","d"))))))
res2: Int = 2
```

Exercise 3.3.5.3* Define a tail-recursive function `foldLeft` for a regular-shaped binary tree. The required type signature and a test:

```
@tailrec def foldLeft[A, R](t: RTree[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(Branch(Leaf(((1,2),(3,4))))))(0)(_ + _)
res0: Int = 10

scala> foldLeft(Branch(Branch(Leaf((("a","b"),("c","d"))))))("")(_ + _)
res1: String = abcd
```

3.3.6 Abstract syntax trees

Expressions in formal languages are represented by abstract syntax trees. An **abstract syntax tree** (or **AST** for short) is defined as either a leaf of one of the available leaf types, or a branch of one of the available branch types. All the available leaf and branch types must be specified as part of the definition of an AST. In other words, one must specify the data carried by leaves and branches, as well as the branching numbers.

To illustrate how ASTs are used, let us rewrite Example 3.2.2.4 via an AST. We view Example 3.2.2.4 as a small sub-language that deals with "safe integers" and supports the "safe arithmetic" operations `Sqrt`, `Add`, `Mul`, and `Div`. Example calculations in this sub-language are $\sqrt{16} * (1 + 2) = 12$; $20 + 1/0 =$ error; and $10 + \sqrt{-1} =$ error.

We can implement this sub-language in two stages. The first stage will create a data structure (an AST) that represents an unevaluated expression in the sub-language. The second stage will evaluate that AST to obtain either a number or an error message.

A straightforward way of defining a data structure for an AST is to use a disjunctive type whose cases describe all the possible operations of the sub-language. We will need one case class for each of `Sqrt`, `Add`, `Mul`, and `Div`. An additional operation, `Num`, will lift ordinary integers into "safe integers". So, we define the disjunctive type for "arithmetic sub-language expressions" as

```
sealed trait Arith
final case class Num(x: Int) extends Arith
final case class Sqrt(x: Arith) extends Arith
final case class Add(x: Arith, y: Arith) extends Arith
final case class Mul(x: Arith, y: Arith) extends Arith
final case class Div(x: Arith, y: Arith) extends Arith
```

A value of type `Arith` is either a `Num(x)` for some integer `x`, or an `Add(x, y)` where `x` and `y` are previously defined `Arith` expressions, or another operation.

This type definition is similar to the binary tree type

```
sealed trait Tree
final case class Leaf(x: Int) extends Tree
final case class Branch(x: Tree, y: Tree) extends Tree
```

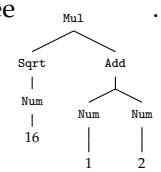
if we rename `Leaf` to `Num` and `Branch` to `Add`. However, the `Arith` type contains 4 different types of "branches", some with branching number 1 and others with branching number 2.

This example illustrates the structure of an AST: it is a tree of a general shape, where leaves and branches are chosen from a specified set of allowed possibilities. In this example, we have a single allowed type of leaf (`Num`) and 4 allowed types of branches (`Sqrt`, `Add`, `Mul`, and `Div`).

This completes the first stage of implementing the sub-language of "safe arithmetic". Using the definition of the disjunctive type `Arith`, we may now create expressions in the sub-language. For example, $\sqrt{16} * (1 + 2)$ is represented by

```
scala> val x: Arith = Mul(Sqrt(Num(16)), Add(Num(1), Num(2)))
x: Arith = Mul(Sqrt(Num(16)),Add(Num(1),Num(2)))
```

We can visualize `x` as the abstract syntax tree



.

The expressions $20 + 1/0$ and $10 * \sqrt{-1}$ are represented by

```
scala> val y: Arith = Add(Num(20), Div(Num(1), Num(0)))
y: Arith = Add(Num(20),Div(Num(1),Num(0)))

scala> val z: Arith = Add(Num(10), Sqrt(Num(-1)))
z: Arith = Add(Num(10),Sqrt(Num(-1)))
```

As we see, the expressions `x`, `y`, and `z` *remain unevaluated*; they are data structures that encode a tree of operations of the sub-language. These operations will be evaluated at the second stage of implementing the sub-language.

To evaluate the expressions in the "safe arithmetic", we can write a function `run: Arith => Either[String, Int]`. That function plays the role of an **interpreter** or "**runner**" for programs written in the sub-language. The runner will destructure the expression tree and execute all the operations, taking care of possible errors.

To implement `run`, we need to define the "safe arithmetic" operations for the result type `Either[String, Int]`. Instead of custom code from Example 3.2.2.4, we can use the `.map` and `.flatMap` operations defined for the `Either` type. For example, addition and multiplication of two "safe integers" is written as

```
def add(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 + r2) }
def mul(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 * r2) }
```

while the "safe division" is

```
def div(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.flatMap(r2 =>
  if (r2 == 0) Left(s"error: $r1 / $r2") else Right(r1 / r2) )
}
```

With this code, we can implement the runner as

```
def run: Arith => Either[String, Int] = {
  case Num(x)     => Right(x)
  case Sqrt(x)    => run(x).flatMap { r =>
   if (r < 0) Left(s"error: sqrt($r)") else
       Right(math.sqrt(r).toInt)
  }
  case Add(x, y)  => add(run(x), run(y))
  case Mul(x, y)  => mul(run(x), run(y))
  case Div(x, y)  => div(run(x), run(y))
}
```

Test with the values `x`, `y`, `z` defined previously:

```
scala> run(x)
res0: Either[String, Int] = Right(12)

scala> run(y)
res1: Either[String, Int] = Left("error: 1 / 0")

scala> run(z)
res2: Either[String, Int] = Left("error: sqrt(-1)")
```

3.4 Summary

What problems can we solve now?

- Represent values from disjoint domains as a single disjunctive type.
- Use disjunctive types instead of exceptions to indicate failures.
- Use disjunctive types to define and work with lists and trees.

The following examples and exercises illustrate the use of disjunctive types.

3.4.1 Solved examples

Example 3.4.1.1 Define a disjunctive type `DayOfWeek` representing the seven days.

Solution Since there is no information other than the label on each day, we use empty case classes:

```
sealed trait DayOfWeek
final case class Sunday() extends DayOfWeek
final case class Monday() extends DayOfWeek
final case class Tuesday() extends DayOfWeek
final case class Wednesday() extends DayOfWeek
final case class Thursday() extends DayOfWeek
final case class Friday() extends DayOfWeek
final case class Saturday() extends DayOfWeek
```

Example 3.4.1.2 Modify `DayOfWeek` so that the values additionally represent a restaurant name and total amount for Fridays and a wake-up time on Saturdays.

Solution For the days where additional information is given, we use non-empty case classes:

```
sealed trait DayOfWeekX
final case class Sunday() extends DayOfWeekX
final case class Monday() extends DayOfWeekX
final case class Tuesday() extends DayOfWeekX
final case class Wednesday() extends DayOfWeekX
final case class Thursday() extends DayOfWeekX
final case class Friday(restaurant: String, amount: Int) extends DayOfWeekX
final case class Saturday(wakeUpAt: java.time.LocalTime) extends DayOfWeekX
```

Example 3.4.1.3 Define a disjunctive type that describes the real roots of the equation $ax^2 + bx + c = 0$, where a, b, c are arbitrary real numbers.

Solution Begin by solving the equation and enumerating all possible cases. It may happen that $a = b = c = 0$, and then all x are roots. If $a = b = 0$ but $c \neq 0$, the equation is $c = 0$, which has no roots. If $a = 0$ but $b \neq 0$, the equation becomes $bx + c = 0$, having a single root. If $a \neq 0$ and $b^2 > 4ac$, we have two distinct real roots. If $a \neq 0$ and $b^2 = 4ac$, we have one real root. If $b^2 < 4ac$, we have no real roots. The resulting type definition can be written as

```
sealed trait RootsOfQ2
final case class AllRoots() extends RootsOfQ2
final case class ConstNoRoots() extends RootsOfQ2
final case class Linear(x: Double) extends RootsOfQ2
final case class NoRealRoots() extends RootsOfQ2
final case class OneRootQ(x: Double) extends RootsOfQ2
final case class TwoRootsQ(x: Double, y: Double) extends RootsOfQ2
```

This disjunctive type contains six parts, among which three parts are empty tuples and two parts are single-element tuples; but this is not a useless redundancy. We would lose information if we reuse `Linear` for the two cases $a = 0$, $b \neq 0$ and $a \neq 0$, $b^2 = 4ac$, or if we reuse `NoRoots()` for representing all three different no-roots cases.

Example 3.4.1.4 Define a function `rootAverage` that computes the average value of all real roots of a general quadratic equation, where the roots are represented by the type `RootsOfQ2` defined in Example 3.4.1.3. The required type signature is

```
val rootAverage: RootsOfQ2 => Option[Double] = ???
```

The function should return `None` if the average is undefined.

Solution The average is defined only in cases `Linear`, `OneRootQ`, and `TwoRootsQ`. In all other cases, we must return `None`. We implement this via pattern matching:

```
val rootAverage: RootsOfQ2 => Option[Double] = { roots =>
  roots match {
    case Linear(x)       => Some(x)
    case OneRootQ(x)     => Some(x)
    case TwoRootsQ(x, y) => Some((x + y) * 0.5)
    case _               => None
  }
}
```

We do not need to enumerate all other cases since the underscore (_) matches everything that the previous cases did not match.

The often-used code pattern of the form `x => x match { case ... }` can be shortened to the nameless function syntax `{ case ...}`. The code then becomes

```
val rootAverage: RootsOfQ2 => Option[Double] = {
  case Linear(x)       => Some(x)
  case OneRootQ(x)     => Some(x)
  case TwoRootsQ(x, y) => Some((x + y) * 0.5)
  case _               => None
}
```

Test it:

```
scala> Seq(NoRealRoots(),OneRootQ(1.0), TwoRootsQ(1.0, 2.0), AllRoots()).
  map(rootAverage)
res0: Seq[Option[Double]] = List(None, Some(1.0), Some(1.5), None)
```

Example 3.4.1.5 Generate 100 quadratic equations $x^2 + bx + c = 0$ with random coefficients b, c (uniformly distributed between −1 and 1) and compute the mean of the largest real roots from all these equations.

Solution We use the type `QEqu` and the `solve` function from Example 3.2.2.1. Create a sequence of equations with random coefficients via the method `Seq.fill`:

```
def random(): Double = scala.util.Random.nextDouble() * 2 - 1
val coeffs: Seq[QEqu] = Seq.fill(100)(QEqu(random(), random()))
```

Then use the `solve` function to compute all roots:

```
val solutions: Seq[RootsOfQ] = coeffs.map(solve)
```

For each set of roots, compute the largest root:

```
scala> val largest: Seq[Option[Double]] = solutions.map {
  case OneRoot(x)     => Some(x)
  case TwoRoots(x, y) => Some(math.max(x, y))
  case _              => None
}
largest: Seq[Option[Double]] = List(None, Some(0.9346072365885472), Some(1.1356234869160806),
    Some(0.9453181931646322), Some(1.1595052441078866), None, Some(0.5762252742788)...
```

It remains to remove the `None` values and to compute the mean of the resulting sequence. The Scala library defines the `.flatten` method that removes `None`s and transforms `Seq[Option[A]]` into `Seq[A]`:

```
scala> largest.flatten
res0: Seq[Double] = List(0.9346072365885472, 1.1356234869160806, 0.9453181931646322,
    1.1595052441078866, 0.5762252742788...
```

Now we can compute the mean of the last sequence. Since the `.flatten` operation is preceded by `.map`, we can replace it by a `.flatMap`. The final code is

```
val largest = Seq.fill(100)(QEqu(random(), random()))
  .map(solve)
  .flatMap {
    case OneRoot(x)     => Some(x)
    case TwoRoots(x, y) => Some(math.max(x, y))
    case _              => None
  }

scala> largest.sum / largest.size
res1: Double = 0.7682649774589514
```

Example 3.4.1.6 Implement a function with type signature

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = ???
```

The function should preserve as much information as possible.

Solution Begin by pattern matching on the argument:

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
  case None                    => ???
  case Some(eab:Either[A, B])  => ???
}
```

The **type annotation** `:Either[A, B]` was written only for clarity; it is not required here because the Scala compiler can deduce the type of the pattern variable `eab` from the fact that we are matching a value of type `Option[Either[A, B]]`.

In the scope of `case None => ???`, we need to return a value of type `Either[A, Option[B]]`. A value of that type must be either a `Left(x)` for some `x:A`, or a `Right(y)` for some `y:Option[B]`, where `y` must be either `None` or `Some(z)` with a `z:B`. However, in our case the code is of the form `case None => ???`, and we cannot produce any values `x:A` or `z:B` since `A` and `B` are arbitrary, unknown types. The only remaining possibility is to return `Right(y)` with `y = None`, and so the code must be

```
  ...
  case None => Right(None) // No other choice here.
```

In the next scope, we can perform pattern matching on the value `eab`:

```
  ...
  case Some(eab: Either[A, B]) = eab match {
    case Left(a)   => ???
    case Right(b)  => ???
  }
```

It remains to figure out what expressions to compute in each case. In the case `Left(a) => ???`, we have a value of type `A`, and we need to compute a value of type `Either[A, Option[B]]`. We execute the same argument as before: The return value must be `Left(x)` for some `x:A`, or `Right(y)` for some `y:Option[B]`. At this point, we have a value of type `A` but no values of type `B`. So we have two possibilities: to return `Left(a)` or to return `Right(None)`. If we decide to return `Left(a)`, the code is

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
  case None       => Right(None) // No other choice here.
  case Some(eab)  => eab match {
    case Left(a)  => Left(a)   // Could also return Right(None) here.
    case Right(b) => ???
  }
}
```

Let us consider the choice of whether to return `Left(a)` or `Right(None)` in the line `case Left(a) => ...`. Both choices will satisfy the required return type `Either[A, Option[B]]`. However, if we return `Right(None)` in that line, we will ignore the given value `a:A`, which loses information. So we return `Left(a)` in that line.

Reasoning similarly for the last line `case Right(b) => ???`, we find that we have a choice of returning `Right(None)` or `Right(Some(b))`. The first choice ignores the given value of `b:B`. To preserve information, we make the second choice:

```
def f1[A, B]: Option[Either[A, B]] => Either[A, Option[B]] = {
  case None       => Right(None)
  case Some(eab)  => eab match {
      case Left(a)    => Left(a)
      case Right(b)   => Right(Some(b))
  }
}
```

Example 3.4.1.7 Implement a function with the type signature

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = ???
```

The function should preserve as much information as possible.

Solution Begin by pattern matching on the argument:

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
  case (Some(a), Some(b)) => ???
  case ???                => ???
```

In the first case, we have values `a:A` and `b:B`, and we are required to return a value of type `Option[(A, B)]`. A value of that type is either `None` or `Some((x, y))` where `x:A` and `y:B`. Since `A` and `B` are arbitrary types, we cannot produce new values `x` and `y` from scratch. The only way to satisfy the required type is to set `x = a` and `y = b`, returning `Some((a, b))`. Now we have two choices: to return `Some((a, b))` or to return `None`. Returning `None` would unnecessarily lose information, so we write

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
  case (Some(a), Some(b)) => Some((a, b))
  case (Some(a), None)    => ???
  ...
```

In the branch of `case (Some(a), None)`, we have a value `a:A` but no values of type `B`. Since the type `B` is arbitrary, we cannot produce any values of type `B` to return a value of the form `Some((x, y))`. So, in this `case` branch, the only computable value of type `Option[(A, B)]` is `None`. We continue to write the code of the function:

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
  case (Some(a), Some(b)) => Some((a, b))
  case (Some(a), None)    => None // No other choice here.
  case (None, Some(b))    => ???
  case (None, None)       => ???
}
```

Writing out the remaining cases, we find that in all those cases we have no choice other than returning `None`. So we can simplify the code:

```
def f2[A, B]: (Option[A], Option[B]) => Option[(A, B)] = {
  case (Some(a), Some(b)) => Some((a, b))
  case _                  => None // No other choice here.
}
```

3.4.2 Exercises

Exercise 3.4.2.1 Define a disjunctive type `CellState` representing the visual state of one cell in the "Minesweeper" game: A cell can be closed (showing nothing), or show a bomb, or be open and show the number of bombs in neighbor cells.

Exercise 3.4.2.2 Define a function from `Seq[Seq[CellState]]` to `Int`, counting the total number of cells with zero neighbor bombs shown.

Exercise 3.4.2.3 Define a disjunctive type `RootOfLinear` representing all possibilities for the solution of the equation $ax + b = 0$ for arbitrary real a, b. (The possibilities are: no roots; one root; all x are roots.) Implement the solution as a function `solve1` with type signature

```
def solve1: ((Double, Double)) => RootOfLinear = ???
```

Exercise 3.4.2.4 Given a `Seq[(Double, Double)]` containing pairs (a, b) of the coefficients of $ax + b = 0$, produce a `Seq[Double]` containing the roots of that equation when a unique root exists. Use the type `RootOfLinear` and the function `solve1` defined in Exercise 3.4.2.3.

Exercise 3.4.2.5 The case class `Subscriber` was defined in Example 3.2.3.1. Given a `Seq[Subscriber]`, compute the sequence of email addresses for all subscribers that did *not* provide a phone number.

Exercise 3.4.2.6 In this exercise, a "procedure" is a function of type `Unit => Unit`; an example of a procedure is `{() => println("hello")}`. Define a disjunctive type `Proc` for an abstract syntax tree representing three operations on procedures: 1) `Func[A]`, creating a procedure from a function of type `Unit => A`, where `A` is a type parameter. 2) `Sequ(p1, p2)`, executing two procedures sequentially. 3) `Para(p1, p2)`, executing two procedures in parallel. Then implement a "runner" that converts a `Proc` into a `Future[Unit]`, running the computations either sequentially or in parallel as appropriate. Test with this code:

```
sealed trait Proc; final case class ??? // etc.
def runner: Proc => Future[Unit] = ???
val proc1: Proc = Func{_ => Thread.sleep(200); println("hello1")}
val proc2: Proc = Func{_ => Thread.sleep(400); println("hello2")}

scala> runner(Sequ(Para(proc2, proc1), proc2))
hello1
hello2
```

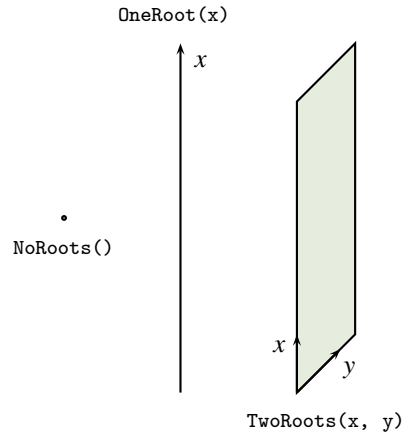


Figure 3.1: The disjoint domain represented by the `RootsOfQ` type.

```
hello2
```

Exercise 3.4.2.7 Implement functions that have a given type signature and preserve as much information as possible:

```
def f1[A, B]: Option[(A, B)] => (Option[A], Option[B]) = ???
def f2[A, B]: Either[A, B] => (Option[A], Option[B]) = ???
def f3[A,B,C]: Either[A, Either[B,C]] => Either[Either[A,B], C] = ???
```

3.5 Discussion

3.5.1 Disjunctive types as mathematical sets

To understand the properties of disjunctive types from the mathematical point of view, consider a function whose argument is a disjunctive type, such as

```
def isDoubleRoot(r: RootsOfQ) = ...
```

The type of the argument `r:RootsOfQ` represents the mathematical domain of the function, that is, the set of admissible values of the argument `r`. We could imagine a function on a *disjoint* domain, for example a domain consisting of a line and a surface, where the surface and the line do not intersect (have no common points). Such domains are called **disjoint**.

The set of real roots of a quadratic equation $x^2 + bx + c = 0$ is an example of a disjoint domain containing three connected parts: the no-roots case, the one-root case where the root is represented by a single number x, and the two-roots case where the roots are represented by a pair of numbers (x, y). Geometrically, a number x is pictured as a point on a line (a one-dimensional space), and pair of numbers (x, y) is pictured as a point on a Cartesian plane (a two-dimensional space). The no-roots case corresponds to a zero-dimensional space, which is pictured as a single point (see Figure 3.1).

In the mathematical notation, a one-dimensional real space is denoted by $\mathbb{R}$, a two-dimensional space by $\mathbb{R}^2$, and a zero-dimensional space by $\mathbb{R}^0$.

At first sight, we may think that the mathematical representation of the type `RootsOfQ` is a union of the three sets, $\mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2$. But an ordinary union of sets would not work for two reasons. First, usually a point is considered as a subset of a line, and a line as a subset of a plane. This amounts to the assumption $\mathbb{R}^0 \subset \mathbb{R}^1 \subset \mathbb{R}^2$, which means $\mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 = \mathbb{R}^2$. If we do not make the assumption $\mathbb{R}^0 \subset \mathbb{R}^1 \subset \mathbb{R}^2$, we encounter the second problem: the need to distinguish the parts of the union unambiguously, even if some parts have the same type. The disjunctive type shown in Example 3.4.1.3 cannot be correctly represented by the mathematical union

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2$$

because $\mathbb{R}^0 \cup \mathbb{R}^0 = \mathbb{R}^0$ and $\mathbb{R}^1 \cup \mathbb{R}^1 = \mathbb{R}^1$, so

$$\mathbb{R}^0 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 = \mathbb{R}^0 \cup \mathbb{R}^1 \cup \mathbb{R}^2 \quad ,$$

but this is not the set we need.

In the Scala code, each part of a disjunctive type must be distinguished by a unique name such as `NoRoots`, `OneRoot`, and `TwoRoots`. To represent this mathematically, we can attach a distinct label to each part of the union. Labels are symbols without any special meaning, and we can just assume that labels are names of Scala case classes. Parts of the union are then represented by sets of pairs such as $(\texttt{OneRoot}, x)_{x \in \mathbb{R}^1}$. Then the domain `RootsOfQ` is expressed as

$$\texttt{RootsOfQ} = (\texttt{NoRoots}, u)_{u \in \mathbb{R}^0} \cup (\texttt{OneRoot}, x)_{x \in \mathbb{R}^1} \cup (\texttt{TwoRoots}, (x, y))_{(x,y) \in \mathbb{R}^2} \quad .$$

This is an ordinary union of mathematical sets, but each of the sets has a unique label, so no two values from different parts of the union could possibly be equal. This kind of labeled union is called a **disjoint union**. Each element of the disjoint union is a pair of the form `(label, data)`, where the label uniquely identifies the part of the union, and the data can have any chosen type such as $\mathbb{R}^1$. If we use disjoint unions, we cannot confuse different parts of the union even if their data have the same type, because labels are required to be distinct.

Disjoint unions are not often explicitly denoted in mathematics, but they are needed in software engineering because real-life data often has that form.

Named `Unit` types At first sight, it may seem strange that the zero-dimensional space is represented by a set containing *one* point. Why should we not use an empty set (rather than a set with one point) to represent the case where the equation has no real roots? The reason is that we are required to represent not only the values of the roots but also the information *about* the existence of the roots. The case with no real roots needs to be represented by some *value* of type `RootsOfQ`. This value cannot be missing, which would happen if we used an empty set to represent the no-roots case. It is natural to use the named empty tuple `NoRoots()` to represent this case, since we used a named 2-tuple `TwoRoots(x, y)` to represent the case of two roots.

Consider the value u used by the mathematical set $(\texttt{NoRoots}, u)_{u \in \mathbb{R}^0}$. Since $\mathbb{R}^0$ consists of a single point, there is only one possible value of u. Similarly, the `Unit` type in Scala has only one distinct value, written as `()`. A case class with no parts, such as `NoRoots`, has only one distinct value, written as `NoRoots()`. This value is syntactically and semantically analogous to $(\texttt{NoRoots}, u)_{u \in \mathbb{R}^0}$.

We see that case classes with no parts are quite similar to `Unit` except for an added name. For this reason, they can be viewed as "named `Unit`" types.

3.5.2 Disjunctive types in other programming languages

Disjunctive types and the associated pattern matching turns out to be one of the defining features of functional programming languages. Programming languages that were not designed for functional

programming do not support these features, while ML, OCaml, Haskell, F#, Scala, Swift, Elm, and PureScript support disjunctive types and pattern matching as part of the language design.

It is remarkable that the named tuple types (also called "structs" or "records") are provided in almost every programming language, while disjunctive types are almost never present except in languages designed for the FP paradigm.[2]

The `union` types in C and C++ are not disjunctive types because it is not possible to determine which part of the union is being represented by a given value. A `union` declaration in C looks like this,

```
union { int x; double y; long z; } di;
```

The problem is that we cannot determine whether a given value `di` represents an `int`, a `double`, or a `long`. This leads to errors that are hard to detect.

Programming languages of the C family (C, C++, Objective C, Java) have "enumeration" (`enum`) types, which *are* a limited form of disjunctive types. An `enum` type declaration in Java looks like this:

```
enum Color { RED, GREEN, BLUE; }
```

In Scala, this is equivalent to a disjunctive type containing three *empty* tuples,

```
sealed trait Color
final case class RED() extends Color
final case class GREEN() extends Color
final case class BLUE() extends Color
```

If the `enum` construction were "enriched" with extra data, so that each tuple could be non-empty, we would obtain the full functionality of disjunctive types. A definition of `RootsOfQ` could then look like this:

```
// This is not valid in Java!
enum RootsOfQ {
  NoRoots(), OneRoot(x: Double), TwoRoots(x: Double, y: Double);
}
```

A future version of Scala 3 will have a shorter syntax for disjunctive types[3] that indeed looks like an "enriched `enum`",

```
enum RootsOfQ {
  case NoRoots
  case OneRoot(x: Double)
  case TwoRoots(x: Double, y: Double)
}
```

For comparison, here is the OCaml and the Haskell syntax for a disjunctive type equivalent to `RootsOfQ`:

```
{* OCaml *}
type RootsOfQ = NoRoots | OneRoot of float | TwoRoots of float*float

-- Haskell
data RootsOfQ = NoRoots | OneRoot Double | TwoRoots (Double,Double)
```

This syntax is more concise than the Scala syntax. When reasoning about disjunctive types, it is inconvenient to write out long type definitions. Chapter 5 will define a short mathematical notation designed for efficient reasoning about types and code.

[2] See this Wikipedia page for a detailed comparison between programming languages. Ada and Pascal are the only languages that have disjunctive types without other FP features.

[3] See https://dotty.epfl.ch/docs/reference/enums/adts.html for details.

3.5.3 Disjunctions and conjunctions in formal logic

In logic, a **proposition** is a logical formula that could be true or false. A **disjunction** of propositions A, B, C is denoted by $A \vee B \vee C$ and is true if and only if *at least one* of A, B, C is true. A **conjunction** of A, B, C is denoted by $A \wedge B \wedge C$ and is true if and only if *all* of the propositions A, B, C are true.

There is a similarity between a disjunctive data type and a logical disjunction of propositions. A value of the disjunctive data type `RootsOfQ` can be constructed only if we have one of the values `NoRoots()`, `OneRoot(x)`, or `TwoRoots(x, y)` for some specific `x` and `y`. Let us now rewrite the previous sentence as a logical formula. Denote by $\mathcal{CH}(A)$ the logical proposition "this $\mathcal{C}$ode $\mathcal{H}$as a value of type `A`", where "this code" refers to a particular expression or function in our program. So, the proposition "a function *can* return a value of type `RootsOfQ`" is denoted by $\mathcal{CH}(\texttt{RootsOfQ})$. We can then write the above sentence about `RootsOfQ` as the logical formula

$$\mathcal{CH}(\texttt{RootsOfQ}) = \mathcal{CH}(\texttt{NoRoots}) \vee \mathcal{CH}(\texttt{OneRoot}) \vee \mathcal{CH}(\texttt{TwoRoots}) \quad . \tag{3.1}$$

There is also a a similarity between logical *conjunctions* and named tuple types. Consider the named tuple `TwoRoots(x: Double, y: Double)`. When can we have a value of type `TwoRoots`? Only when we have two values of type `Double`. Rewriting this sentence as a logical formula, we get

$$\mathcal{CH}(\texttt{TwoRoots}) = \mathcal{CH}(\texttt{Double}) \wedge \mathcal{CH}(\texttt{Double}) \quad .$$

Now, formal logic admits the simplification

$$\mathcal{CH}(\texttt{Double}) \wedge \mathcal{CH}(\texttt{Double}) = \mathcal{CH}(\texttt{Double}) \quad .$$

However, no such simplification will be available in the general case, e.g.

```
case class Data3(x: Int, y: String, z: Double)
```

For this type, we will have the formula

$$\mathcal{CH}(\texttt{Data3}) = \mathcal{CH}(\texttt{Int}) \wedge \mathcal{CH}(\texttt{String}) \wedge \mathcal{CH}(\texttt{Double}) \quad . \tag{3.2}$$

We find that tuples are related to logical conjunctions in the same way as disjunctive types are related to logical disjunctions. This is the main motivation for choosing the name "disjunctive types".[4]

The correspondence between disjunctions, conjunctions, and data types is explained in more detail in Chapter 5. For now, we note that the operations of conjunction and disjunction are not sufficient to produce all possible logical expressions. To obtain a complete logic, it is also necessary to have a logical negation $\neg A$ ("A is not true") or, equivalently, a logical implication $A \Rightarrow B$ ("if A is true than B is true"). It turns out that the logical implication $A \Rightarrow B$ is related to the function type `A => B`. In Chapter 4, we will study function types in depth.

[4] These types are also called "variants", "sum types", "co-product types", and "tagged union types".

Part II

Intermediate level

4 The logic of types. II. Higher-order functions

4.1 Functions that return functions

4.1.1 Motivation and a first example

Consider the task of preparing a logger function that prints messages but adds a fixed prefix to each message.

A simple logger function can be a value of type `String => Unit`, such as

```
val logger: String => Unit = { message => println(s"INFO: $message") }

scala> logger("hello world")
INFO: hello world
```

This function prints any given message with the logging prefix `"INFO"`.

The standard library function `println(...)` always returns a `Unit` value after printing its arguments. As we already know, there is only a single value of type `Unit`, and that value is denoted by `()`. To verify, run this code:

```
scala> val x = println(123)
123
x: Unit = ()
```

The task is to make the logging prefix configurable. A simple solution is to implement a function `logWith` that takes a prefix as an argument and returns a new logger with that prefix fixed. It is important that the function `logWith` will return a new value of type `String => Unit`, i.e. a new *function*:

```
def logWith(prefix: String): (String => Unit) = {
    message => println(s"$prefix: $message")
}
```

The body of `logWith` consists of a nameless function `message => println(...)`, which is a value of type `String => Unit`. This value will be computed by `logWith("...")`.

We can now use `logWith` to create a few logger functions:

```
scala> val info = logWith("INFO")
info: String => Unit = <function1>

scala> val warn = logWith("WARN")
warn: String => Unit = <function1>
```

The created logger functions are then used as ordinary functions:

```
scala> info("hello")
INFO: hello

scala> warn("goodbye")
WARN: goodbye
```

The values `info` and `warn` can be used by any code that needs a logging function.

It is important that the prefix is "baked into" functions created by `logWith`. A logger such as `warn` will always print messages with the prefix `"WARN"`, and the prefix cannot be changed any more. This is so because the value `prefix` is treated as a local value in the scope of the function returned by `logWith`. For instance, the body of the function `warn` is equivalent to

```
{ val prefix = "WARN"; (message => s"$prefix: $message") }
```

So, whenever a new function is created using `logWith(prefix)`, the (immutable) value of `prefix` is stored within the body of the newly created function. This is a general feature of nameless functions created in a scope that contains other local values: the function body keeps a copy of all the local values it uses. One sometimes says that the function body "closes over" the local values; for this reason, nameless functions are sometimes also called "**closure**s". It would be perhaps clearer to say that nameless functions "capture" local values.

As another example of the capture of local values, consider this code:

```
val f: Int => Int = {
  val p = 10
  val q = 20
  x => p + q * x
}
```

The body of the function `f` is equivalent to `{x => 10 + 20 * x}` because the values `p = 10` and `q = 20` were captured.

4.1.2 Curried and uncurried functions

Reasoning mathematically about the code

```
val info = logWith("INFO")
info("hello")
```

we would expect that `info` is *the same value* as `logWith("INFO")`, and so the code `info("hello")` should have the same effect as the code `logWith("INFO")("hello")`. This is indeed so:

```
scala> logWith("INFO")("hello")
INFO: hello
```

The syntax `logWith("INFO")("hello")` looks like the function `logWith` applied to *two* arguments. Yet, `logWith` was defined as a function with a single argument of type `String`. This is not a contradiction because `logWith("INFO")` returns a function that accepts an additional argument. So, both function applications `logWith("INFO")` and `logWith("INFO")("hello")` are valid. In this sense, we are allowed to apply `logWith` to one argument at a time.

A function that can be applied to more than one argument in this way is called a **curried** function. A curried function can be applied to one argument at a time.

An **uncurried** function must be applied to all arguments at once, e.g.

```
def prefixLog(prefix: String, message: String): Unit =
  println(s"$prefix: $message")
```

and returns a value of a non-function type.

The type of the curried function `logWith` is `String => (String => Unit)`. Scala adopts the syntax convention that the function arrow (`=>`) groups to the *right*. So the parentheses in the type expression `String => (String => Unit)` are not necessary; the function's type can be written as `String => String => Unit`.

The type `String => String => Unit` is different from `(String => String) => Unit`, – the type of a function returning `Unit` and taking a function of type `String => String` as its argument. When an argument's type is a function type, e.g. `String => String`, it must be enclosed in parentheses.

In general, a curried function takes an argument and returns another function that again takes an argument and returns another function, and so on, until finally a non-function type is returned. So, the type signature of a curried function generally looks like `A => B => C => ... => R => S`, where `A`, `B`, ..., `R` are the **curried arguments** and `S` is the "final" result type.

For most people, it takes time to get used to reading this kind of syntax. In the type expression `A => B => C => D`, the first three types are curried arguments and the type `D` is the final result type.

In Scala, functions defined with multiple argument groups (enclosed in multiple pairs of parentheses) are curried functions. We have seen examples of curried functions before:

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B]
def fmap[A, B](f: A => B)(xs: Option[A]): Option[B]
def foldLeft[A, R](xs: Seq[A])(init: R)(update: (R, A) => R): R
```

The type signatures of these functions can be also written equivalently as

```
def map[A, B]: Seq[A] => (A => B) => Seq[B]
def fmap[A, B]: (A => B) => Option[A] => Option[B]
def foldLeft[A, R]: Seq[A] => R => ((R, A) => R) => R
```

Curried arguments of a function type, such as `(A => B)`, need parentheses.

The curried function `logWith` can be defined in three equivalent ways in Scala:

```
def logWith1(prefix: String)(message: String): Unit =
  println(s"$prefix: $message")
def logWith2(prefix: String): String => Unit =
  message => println(s"$prefix: $message")
def logWith3: String => String => Unit =
  prefix => message => println(s"$prefix: $message")
```

Nameless functions in Scala may be enclosed in parentheses or curly braces. We will omit parentheses for simple expressions.

The last line in the above code shows that the arrow `=>` groups to the right within the *code* of nameless functions: `x => y => expr` means `{x => {y => expr}}`, a nameless function taking an argument `x` and returning a nameless function that takes an argument `y` and returns an expression `expr`. This syntax convention is useful for two reasons. First, the code `x => y => z` visually corresponds to the curried function's type signature `A => B => C`, which uses the same syntax convention. Second, the syntax `(x => y) => z` could not be valid for a nameless function because `(x => y)` is not a valid pattern expression for the function's argument; it is impossible to define a pattern that matches arbitrary *functions* of type `A => B`.

Although the code syntax `(x => y) => z` is invalid, a type expression `(A => B) => C` is valid. A nameless function of type `(A => B) => C` is written as `f => expr(f)` where `f: A => B` is its argument and `expr(f)` its body. So `x => (y => z)` is the only possible way of inserting parentheses into `x => y => z`.

4.1.3 Equivalence of curried and uncurried functions

We defined the curried function `logWith` in order to be able to create logger functions such as `info` and `warn`. However, some curried functions, such as `foldLeft`, are almost always applied to all possible arguments. A curried function applied to all its possible arguments is equivalent to an uncurried function that takes all those arguments at once. Let us look at this equivalence in more detail.

Consider a curried function with type signature `Int => Int => Int`. This function takes an integer and returns an (uncurried) function taking an integer and returning an integer. An example of such a curried function is

```
def f1(x: Int): Int => Int = { y => x - y }
```

The function takes an integer `x` and returns the expression `y => x - y`, which is a function of type `Int => Int`. The code of `f1` can be written equivalently as

```
val f1: Int => Int => Int = { x => y => x - y }
```

Let us compare the function `f1` with a function that takes its two arguments at once. Such a function will have a different type signature, e.g.

```
def f2(x: Int, y: Int): Int = x - y
```

has type signature `(Int, Int) => Int` but computes the same value as `f1`.

The syntax for using the functions `f1` and `f2` is different:

```
scala> f1(20)(4)
res0: Int = 16

scala> f2(20, 4)
res1: Int = 16
```

The main difference between the usage of `f1` and `f2` is that `f2` must be applied *at once* to both arguments, while `f1` applied to just the first argument, `20`. The result of evaluating `f1(20)` is a function that can be later applied to another argument:

```
scala> val r1 = f1(20)
r1: Int => Int = <function1>

scala> r1(4)
res2: Int = 16
```

Applying a curried function to some but not all of possible arguments is called a **partial application**. Applying a curried function to all possible arguments is called a **saturated** application.

If we need to apply an *uncurried* function to some of its arguments but leave other arguments unspecified, we can use the underscore (_) symbol:

```
scala> val r2: Int => Int = f2(20, _)
r2: Int => Int = <function1>

scala> r2(4)
res3: Int = 16
```

(Here, the type annotation `Int => Int` is required.) This code creates a function `r2` by partially applying `f2` to the first argument but not to the second. Other than that, `r2` is the same function as `r1` defined above; i.e. `r2` returns the same values for the same arguments as `r1`. A more general syntax for a partial application is

```
scala> val r3: Int => Int = { x => f2(20, x) }
r3: Int => Int = <function1>

scala> r3(4)
res4: Int = 16
```

We can see that a curried function, such as `f1`, is better adapted for partial application than `f2`, because the syntax is shorter. However, the functions `f1` and `f2` are **computationally equivalent** in the sense that given `f1` we can reconstruct `f2` and vice versa:

```
def f2new(x: Int, y: Int): Int = f1(x)(y)
def f1new: Int => Int => Int = { x => y => f2(x, y) }
```

It is clear that the function `f1new` computes the same results as `f1`, and that the function `f2new` computes the same results as `f2`. The computational equivalence of the functions `f1` and `f2` is not *equality* – these functions are *different*; but one of them can be easily reconstructed from the other if necessary.

More generally, a curried function has a type signature of the form `A => B => C => ... => R => S`, where `A`, `B`, `C`, ..., `S` are some types. A function with this type signature is computationally equivalent to an uncurried function with type signature `(A,B,C,...,R) => S`. The uncurried function takes all arguments at once, while the curried function takes one argument at a time. Other than that, these two functions compute the same results given the same arguments.

We have seen how a curried function can be converted to an equivalent uncurried one, and vice versa. The Scala library defines the methods `curried` and `uncurried` that convert between these computationally equivalent forms of functions. Here we convert the function `f2` to `f1` and back:

```
scala> val f1c = (f2 _).curried
f1c: Int => (Int => Int) = <function1>

scala> val f2u = Function.uncurried(f1c)
f2u: (Int, Int) => Int = <function2>
```

The syntax `(f2 _)` is needed in Scala[1] to convert methods to function values. Recall that Scala has two ways of defining a function: one as a method (defined using `def`), another as a function value (defined using `val`).

The methods `.curried` and `.uncurried` are easy to implement in Scala code, as we will show in Section 4.2.1.

4.2 Fully parametric functions

We have seen that some functions are declared with type parameters, which are set only when the function is applied to specific arguments. Examples of such functions are the `map` and `filter` methods with type signatures

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B]
def filter[A](xs: Seq[A])(p: A => Boolean): Seq[A]
```

Such functions can be applied to arguments of different types without changing the function's code. So, it is better to implement a single function with type parameters instead of several functions with the same code but applied to different types. When we apply the function `map` as `map(xs)(f)` to a specific value `xs` of type, e.g., `Seq[Int]`, and a specific function `f` of type, say, `Int => String`, the Scala compiler will automatically set the type parameters `A = Int` and `B = String` in the definition of `map`. We may also set type parameters explicitly and write, for example, `map[Int, String](xs)(f)`. This syntax shows a certain similarity between type parameters such as `Int`, `String` and "value parameters" (arguments) `xs` and `f`.

In the functions `map` and `filter` as just shown, some types are parameters while others are specific types, such as `Seq` and `Boolean`. It is sometimes possible to replace *all* types in the type signature of a function by type parameters.

A function is **fully parametric** if all its arguments have types described by type parameters, and the code of the function works with type parameters rather than with fixed types such as `Int` or `String`. In other words, fully parametric functions do not use any specific types, such as `Int` or `String`, in their type signature or in their body. When a fully parametric function is applied to an argument of a specific type, the function does not use any information about that type.

What kind of functions are fully parametric? To build an intuition for that, let us compare these two functions having the same type signature:

[1] The extra underscore will become unnecessary in Scala 3.

```
def cos_sin(p: (Double, Double)): (Double, Double) = p match {
  case (x, y) =>
    val r = math.sqrt(x * x + y * y)
    (x / r, y / r) // Return cos and sin of the angle.
}

def swap(p: (Double, Double)): (Double, Double) = p match {
  case (x, y) => (y, x)
}
```

We can introduce type parameters into the type signature of `swap` to make it fully parametric, with no changes in the code of the function:

```
def swap[A, B](p: (A, B)): (B, A) = p match {
  case (x, y) => (y, x)
}
```

Generalizing `swap` to a fully parametric function is possible because the operation of swapping two parts of a tuple `(A, B)` works in the same way for all types `A`, `B`.

In contrast, the function `cos_sin` performs a computation that is specific to the type `Double` and cannot be generalized to an arbitrary type parameter `A` instead of `Double`. So, `cos_sin` cannot be generalized to a fully parametric function.

The `swap` operation for pairs is already defined in the Scala library:

```
scala> (1, "abc").swap
res0: (String, Int) = (abc,1)
```

Other swapping functions can be defined for tuples with more elements, e.g.

```
def swap12[A,B,C]: ((A, B, C)) => (B, A, C) = {
  case (x, y, z) => (y, x, z)
}
```

The Scala syntax requires the double parentheses around tuple types of arguments but not around the tuple type of a function's result. So, the type signature of `cos_sin` may be written as

```
def cos_sin: ((Double, Double)) => (Double, Double)
```

4.2.1 Examples. Function composition

Further examples of fully parametric functions are the identity function, the constant function, the function composition methods, and the curry / uncurry conversions. The identity function (available in the Scala library as `identity[T]`) is

```
def id[T]: T => T = t => t
```

The constant function (available in the Scala library as `Function.const`) takes an argument `c` and returns a new function that always returns `c`:

```
def const[C, X](c: C): X => C = (_ => c)
```

The syntax `_ => c` is used to emphasize that the function ignores its argument.

Function composition Consider two functions `f: Int => Double` and `g: Double => String`. We can apply `f` to an integer argument `x:Int` and get a result `f(x)` of type `Double`. We can then apply `g` to that result and obtain a `String` value `g(f(x))`. The transformation from the original integer `x:Int` to the final `String` value `g(f(x))` can be viewed as a new function of type `Int => String`. That new function

is called the **forward composition** of the two functions `f` and `g`. In Scala, this operation is written as `f andThen g`:

```
val f: Int => Double = x => 5.67 + x
val g: Double => String = x => f"Result x = $x%3.2f"

scala> val h = f andThen g
h: Int => String = <function1>

scala> h(40)
res36: String = Result x = 45.67
```

The Scala compiler derives the type of `h` automatically as `Int => String`.

The forward composition is denoted by $\mathbin{;}$ (pronounced "before") and can be defined as

$$f \mathbin{;} g \triangleq (x \Rightarrow g(f(x))) \quad . \tag{4.1}$$

The symbol $\triangleq$ means "is defined as".

We could write the forward composition as a fully parametric function,

```
def andThen[X, Y, Z](f: X => Y)(g: Y => Z): X => Z = { x => g(f(x)) }
```

The type signature of this curried function is

$$\text{andThen} : (X \Rightarrow Y) \Rightarrow (Y \Rightarrow Z) \Rightarrow X \Rightarrow Z \quad .$$

This type signature requires the types of the function arguments to match in a certain way, or else the composition is undefined.

The method `andThen` is an example of a function that *both* returns a new function *and* takes other functions as arguments.

The **backward composition** of two functions f and g works in the opposite order: first g is applied and then f is applied to the result. Using the symbol $\circ$ (pronounced "after") for this operation, we can write

$$f \circ g \triangleq (x \Rightarrow f(g(x))) \quad . \tag{4.2}$$

In Scala, the backward composition is called `compose` and used as `f compose g`. This method may be implemented as a fully parametric function

```
def compose[X, Y, Z](f: Y => X)(g: Z => Y): Z => X = { z => f(g(z)) }
```

The type signature of this curried function is

$$\text{compose} : (Y \Rightarrow X) \Rightarrow (Z \Rightarrow Y) \Rightarrow Z \Rightarrow X \quad .$$

We have already seen the methods `curried` and `uncurried` defined by the Scala library. As an illustration, let us write our own code for converting curried functions to uncurried:

```
def uncurry[A, B, R](f: A => B => R): ((A, B)) => R = {
  case (a, b) => f(a)(b)
}
```

We conclude from these examples that fully parametric functions perform operations that are so general that they work in the same way for all types of arguments. Some arguments of fully parametric functions may have complicated types such as `A => B => R`, which are type expressions made up from type parameters alone. Fully parametric functions do not perform any operations with specific types such as `Int` or `String`.

The property of being fully parametric is also called **parametricity**. In some programming languages, functions with type parameters are called "generic".

4.2.2 Laws of function composition

The operations of function composition, introduced in Section 4.2.1, have three important properties or "laws" that follow directly from the definitions. These laws are:

- The two identity laws: the composition of any function f with the identity function will give again the function f.
- The associativity law: the consecutive composition of three functions f, g, h does not depend on the order in which the pairs are composed.

These laws hold for the forward and the backward composition, since they are just syntactic variants of the same mathematical operation. Let us write these laws rigorously as mathematical equations and prove them.

Proofs in the forward notation The composition of the identity function with an arbitrary function f can be $\text{id} \fatsemi f$ with the identity function to the left of f, or $f \fatsemi \text{id}$ with the identity function to the right of f. In both cases, the result must be equal to the function f. The resulting two laws are

$$\begin{aligned} \text{left identity law of composition}: \quad & \text{id} \fatsemi f = f \quad , \\ \text{right identity law of composition}: \quad & f \fatsemi \text{id} = f \quad . \end{aligned}$$

To show that these laws always hold, we need to show that both sides of the laws, which are functions, give the same result when applied to an arbitrary value x. Let us first clarify how the type parameters must be set for the laws to have consistent types.

The laws must hold for an arbitrary function f. So we may assume that f has the type signature $A \Rightarrow B$, where A and B are arbitrary type parameters. Consider the left identity law. The function $(\text{id} \fatsemi f)$ is, by definition (4.1), a function that takes an argument x, applies id to that x, and then applies f to the result:

$$\text{id} \fatsemi f = (x \Rightarrow f(\text{id}(x))) \quad .$$

If f has type $A \Rightarrow B$, its argument must be of type A, or else the types will not match. Therefore, the identity function must have type $A \Rightarrow A$, and the argument x must have type A. With these choices of the type parameters, the function $(x \Rightarrow f(\text{id}(x)))$ will have type $A \Rightarrow B$, as it must since the right-hand side of the law is f. We add type annotations to the code as *superscripts*,

$$\text{id}^A \fatsemi f^{:A \Rightarrow B} = (x^{:A} \Rightarrow f(\text{id}(x)))^{:A \Rightarrow B} \quad .$$

In the Scala syntax, this formula may be written as

```
id[A] andThen (f: A => B) == { x: A => f(id(x)) }: A => B
```

It is quicker to write the mathematical notation than code in the Scala syntax. We will follow the convention where type parameters are single uppercase letters; however, this convention is not enforced in Scala.

The colon symbol (:) in the superscript $x^{:A}$ means a type annotation, as in Scala code `x:A`. A superscript without the colon, such as id^A, means a type parameter, as in Scala code `id[A]`. Since the function `id[A]` has type `A => A`, we can write id^A or equivalently (but more verbosely) $\text{id}^{:A \Rightarrow A}$ to denote that function.

Since $\text{id}(x) = x$ by definition of the identity function, we find that

$$\text{id} \fatsemi f = (x \Rightarrow f\,(\text{id}\,(x))) = (x \Rightarrow f\,(x)) = f \quad .$$

The last step works since $x \Rightarrow f\,(x)$ is a function taking an argument x and applying f to that argument; i.e. $x \Rightarrow f\,(x)$ is the same function as f.

Now consider the right identity law:

$$f \fatsemi \mathrm{id} = (x \Rightarrow \mathrm{id}\,(f\,(x))) \quad .$$

To make the types match, assume that $f^{:A \Rightarrow B}$. Then x must have type A, and the identity function must have type $B \Rightarrow B$. The result of $\mathrm{id}\,(f\,(x))$ will be also of type B. With these choices of type parameters, all types match:

$$f^{:A \Rightarrow B} \fatsemi \mathrm{id}^B = \left(x^{:A} \Rightarrow \mathrm{id}(f(x))\right)^{:A \Rightarrow B} \quad .$$

Since $\mathrm{id}(f(x)) = f(x)$, we find that

$$f \fatsemi \mathrm{id} = (x \Rightarrow f\,(x)) = f \quad .$$

In this way, we have demonstrated that both identity laws hold.

The associativity law is written as an equation like this:

$$(f \fatsemi g) \fatsemi h = f \fatsemi (g \fatsemi h) \quad . \tag{4.3}$$

Let us first verify that the types match here. The types of the functions f, g, and h must be such that all the function compositions exist. If f has type $A \Rightarrow B$ for some type parameters A and B, then the argument of g must be of type B; so we can choose $g^{:B \Rightarrow C}$, where C is another type parameter. The composition $f \fatsemi g$ has type $A \Rightarrow C$, so h must be of type $C \Rightarrow D$ for some type D. Assuming the types as $f^{:A \Rightarrow B}$, $g^{:B \Rightarrow C}$, and $h^{:C \Rightarrow D}$, we find that the types in all the compositions $f \fatsemi g$, $g \fatsemi h$, $(f \fatsemi g) \fatsemi h$, and $f \fatsemi (g \fatsemi h)$ match. We can rewrite Eq. (4.3) with type annotations,

$$(f^{:A \Rightarrow B} \fatsemi g^{:B \Rightarrow C}) \fatsemi h^{:C \Rightarrow D} = f^{:A \Rightarrow B} \fatsemi (g^{:B \Rightarrow C} \fatsemi h^{:C \Rightarrow D}) \quad . \tag{4.4}$$

Having checked the types, we are ready to prove the associativity law. We note that both sides of the law (4.4) are functions of type $A \Rightarrow D$. To prove that two functions are equal means to prove that they always return the same results when applied to the same argument. So we need to apply both sides of Eq. (4.4) to an arbitrary value $x^{:A}$. Using the definition (4.1) of the forward composition,

$$(f \fatsemi g)\,(x) = g(f(x)) \quad ,$$

we find

$$\begin{aligned} ((f \fatsemi g) \fatsemi h)\,(x) &= h\,((f \fatsemi g)\,(x)) = h(g(f(x))) \quad , \\ (f \fatsemi (g \fatsemi h))\,(x) &= (g \fatsemi h)\,(f(x)) = h(g(f(x))) \quad . \end{aligned}$$

Both sides of the law are now clearly equal when applied to an arbitrary value x.

Because of the associativity law, we do not need parentheses in the expression $f \fatsemi g \fatsemi h$. The function $(f \fatsemi g) \fatsemi h$ is the same as $f \fatsemi (g \fatsemi h)$.

In the proof, we have omitted the types since we already checked that the types match. Checking the types beforehand allows us to write shorter proofs.

Proofs in the backward notation This book uses the **forward notation** $f \fatsemi g$ for writing function compositions, rather than the backward notation $g \circ f$. If necessary, all equations can be automatically converted from one notation to the other by reversing the order of function compositions, since

$$f \fatsemi g = g \circ f$$

for any functions $f^{:A \Rightarrow B}$ and $g^{:B \Rightarrow C}$. Let us see how to prove the composition laws in the backward notation. We will just need to reverse the order of function compositions in the proofs above.

The left identity and right identity laws are

$$f \circ \mathrm{id} = f \quad , \quad \mathrm{id} \circ f = f \quad .$$

To match the types, we need to choose the type parameters as

$$f^{:A \Rightarrow B} \circ \mathrm{id}^{:A \Rightarrow A} = f^{:A \Rightarrow B} \quad , \quad \mathrm{id}^{B \Rightarrow B} \circ f^{:A \Rightarrow B} = f^{:A \Rightarrow B} \quad .$$

We can apply both sides of the laws to an arbitrary value $x^{:A}$. For the left identity law, we find from definition (4.2) that

$$f \circ \mathrm{id} = (x \Rightarrow f(\mathrm{id}(x))) = (x \Rightarrow f(x)) = f \quad .$$

Similarly for the right identity law,

$$\mathrm{id} \circ f = (x \Rightarrow \mathrm{id}\,(f\,(x))) = (x \Rightarrow f\,(x)) = f \quad .$$

The associativity law,

$$h \circ (g \circ f) = (h \circ g) \circ f \quad ,$$

is proved by applying both sides to an arbitrary value x of a suitable type:

$$(h \circ (g \circ f))\,(x) = h\,((g \circ f)\,(x)) = h\,(g\,(f\,(x))) \quad ,$$
$$((h \circ g) \circ f)\,(x) = (h \circ g)\,(f(x)) = h\,(g\,(f\,(x))) \quad .$$

The types are checked by assuming that f has the type $f^{:A \Rightarrow B}$. The types in $g \circ f$ match only when $g^{:B \Rightarrow C}$, and then $g \circ f$ is of type $A \Rightarrow C$. The type of h must be $h^{:C \Rightarrow D}$ for the types in $h \circ (g \circ f)$ to match. We can write the associativity law with type annotations as

$$h^{:C \Rightarrow D} \circ (g^{:B \Rightarrow C} \circ f^{A \Rightarrow B}) = (h^{:C \Rightarrow D} \circ g^{:B \Rightarrow C}) \circ f^{:A \Rightarrow B} \quad . \tag{4.5}$$

The associativity law allows us to omit parentheses in the expression $h \circ g \circ f$.

The length of calculations is the same in the forward and the backward notation. One difference is that types of function compositions are more visually clear in the forward notation: it is easier to check types in Eq. (4.4) than in Eq. (4.5).

4.2.3 Example: A function that violates parametricity

Fully parametric functions should not make any decisions based on the actual types of its parameters. As an example of an *incorrect* implementation of a fully parametric function, consider the following "fake identity" function:

```
def fakeId[A]: A => A = { // Special code for ‘A‘ = ‘Int‘:
  case x: Int => (x - 1).asInstanceOf[A]
  case x => x // Common code for all other types ‘A‘.
}
```

This function's type signature is the same as that of `id[A]`, and its behavior is the same for all types `A` except for `A = Int`:

```
scala> fakeId("abc")
res0: String = abc

scala> fakeId(true)
res1: Boolean = true

scala> fakeId(0)
res2: Int = -1
```

While Scala allows us to write this kind of code, the resulting function does not appear to be useful. In any case, `fakeId` is not a fully parametric function.

The identity laws of composition will not hold if we use `fakeId[A]` instead of the correct function `id[A]`. For example, consider the composition of `fakeId` with a simple function `f_1` defined by

```
def f_1: Int => Int = { x => x + 1 }
```

The composition (`f_1 andThen fakeId`) will have type `Int => Int`. Since `f_1` has type `Int => Int`, Scala will automatically set the type parameter `A = Int` in `fakeId[A]`,

```
scala> def f_2 = f_1 andThen fakeId
f_2: Int => Int
```

The identity law says that $f_2 = f_1 \fatsemi \mathrm{id} = f_1$. But we can check that `f_1` and `f_2` are not the same:

```
scala> f_1(0)
res3: Int = 1

scala> f_2(0)
res3: Int = 0
```

It is important that we are able to detect a violation of parametricity by checking whether some equation holds, without need to examine the code of the function `fakeId`. In this book, we will always formulate any desired properties of functions through equations or "laws". To prove the laws, we will need to perform symbolic calculations similar to the proofs in Section 4.2.2. These calculations are **symbolic** in the sense that we were manipulating symbols such as x, f, g, and h without substituting any specific values for these symbols but using only the general properties of functions. In the next section, we will get some more experience with such calculations.

4.3 Symbolic calculations with nameless functions

4.3.1 Solved examples: Deriving a function's type from its code

Checking that the types match is an important part of the functional programming paradigm – both in the practice of writing code and in theoretical derivations of laws for various functions. For instance, in the derivations of the composition laws (Section 4.2.2), we were able to deduce the possible type parameters for f, g, and h in the expression $f \fatsemi g \fatsemi h$. This worked because the composition operation `andThen` (denoted by the symbol $\fatsemi$) is fully parametric. Given a fully parametric function, it is often possible to derive the most general type signature that matches the body of that function. The same type-matching procedure may also help in converting a given function to a fully parametric form.

Let us look at some examples of doing this.

Example 4.3.1.1 The functions `const` and `id` were defined in Section 4.2.1. What is the value `const(id)` and what is its type? Determine the most general type parameters in the expression `const(id)`.

Solution We need to treat the functions `const` and `id` as values, since our goal is to apply `const` to `id`. Write the code of these functions in a short notation:

$$\mathrm{const}^{C,X} \triangleq c^{:C} \Rightarrow \_^{:X} \Rightarrow c \quad ,$$
$$\mathrm{id}^{A} \triangleq a^{:A} \Rightarrow a \quad .$$

The types will match in the expression `const(id)` only if the argument of the function `const` has the same type as the type of `id`. Since "const" is a curried function, we need to look at its *first* curried argument, which is of type C. The type of id is $A \Rightarrow A$, where A is an arbitrary type so far. So, the

type parameter C in $\text{const}^{C,X}$ must be equal to $A \Rightarrow A$:

$$C = A \Rightarrow A \quad .$$

The type parameter X in $\text{const}^{C,X}$ is not constrained, so we keep it as X. The result of applying const to id is of type $X \Rightarrow C$, which equals $X \Rightarrow A \Rightarrow A$. In this way, we find that the type of `const(id)`,

$$\text{const}^{A \Rightarrow A, X}(\text{id}^A) : X \Rightarrow A \Rightarrow A \quad .$$

The types A and X can be arbitrary. The type $X \Rightarrow A \Rightarrow A$ is the most general type for the expression `const(id)` because we have not made any assumptions about the types except requiring that all functions must be always applied to arguments of the correct types.

To compute the value of `const(id)`, it remains to substitute the code of `const` and `id`. Since we already checked the types, we may omit all type annotations:

$$\begin{aligned} & \underline{\text{const}}\ (\text{id}) \\ \text{definition of const}: \quad & = (c \Rightarrow x \Rightarrow \underline{c})(\text{id}) \\ \text{substitute } c = \text{id}: \quad & = (x \Rightarrow \underline{\text{id}}) \\ \text{definition of id}: \quad & = (x \Rightarrow a \Rightarrow a) \quad . \end{aligned}$$

This is a function that takes an argument $x^{:X}$ and returns the identity function $a^{:A} \Rightarrow a$. It is clear that the argument x is ignored by this function, so we can rewrite the result equivalently as

$$\text{const(id)} = (\_^{:X} \Rightarrow a^{:A} \Rightarrow a) \quad .$$

Example 4.3.1.2 Implement a function `twice` that takes a function `f: Int => Int` as its argument and returns a function that applies `f` twice. For example, if the function `f` is `f = { x => x + 3 }`, the result of `twice(f)` should be equal to the function `x => x + 6`. After implementing the function `twice`, generalize it to a fully parametric function.

Solution According to the requirements, the function `twice` must return a new function of type `Int => Int`. So the type signature of `twice` is

```
def twice(f: Int => Int): Int => Int = ???
```

Since `twice(f)` must be a new function with an integer argument, we begin to write the body of `twice` as

```
def twice(f: Int => Int): Int => Int = { (x: Int) => ??? }
```

The new function must apply `f` twice to its argument, that is, it must return `f(f(x))`. We can finish the implementation now:

```
def twice(f: Int => Int): Int => Int = { x => f(f(x)) }
```

The type annotation `(x: Int)` can be omitted. To test:

```
scala> val g = twice(x => x + 3)
g: Int => Int = <function1>

scala> g(10)
res0: Int = 16
```

This example illustrates how a function can return a new function. We just write a nameless function in the function body.

To generalize `twice` to a fully parametric function means to replace the type signature by a parameterized type while keeping the function body unchanged,

```
def twice[A, B, ...](f: ...): ... = { x => f(f(x)) }
```

To determine the type signature and the possible type parameters A, B, ..., we need to determine the most general type that matches the function body. The function body is the expression $x \Rightarrow f(f(x))$. Assume that x has type A; for types to match in the sub-expression $f(x)$, we need f to have type $A \Rightarrow B$ for some type B. The sub-expression $f(x)$ will then have type B. For types to match in $f(f(x))$, the argument of f must have type B; but we already assumed $f^{:A \Rightarrow B}$. This is consistent only if $A = B$. In this way, $x^{:A}$ implies $f^{:A \Rightarrow A}$, and the expression $x \Rightarrow f(f(x))$ has type $A \Rightarrow A$. We can now write the type signature of `twice`,

```
def twice[A](f: A => A): A => A = { x => f(f(x)) }
```

This fully parametric function has only one independent type parameter, A, and can be equivalently written in the mathematical notation as the curried function

$$\text{twice}^A \triangleq f^{:A \Rightarrow A} \Rightarrow x^{:A} \Rightarrow f(f(x)) \quad . \tag{4.6}$$

The procedure of deriving the most general type for a given code is called **type inference**. The presence of the type parameter A and the general type signature $(A \Rightarrow A) \Rightarrow A \Rightarrow A$ have been "inferred" from the code $f \Rightarrow x \Rightarrow f(f(x))$.

Example 4.3.1.3 Consider the fully parametric function `twice` defined in Example 4.3.1.2. What is the type of `twice(twice)`, and what computation does it perform? Test your answer on the expression `twice(twice[Int])(x => x+3)(10)`. What are the type parameters in that expression?

Solution Begin by figuring out the required type of `twice(twice)`. We introduce unknown type parameters as `twice[A](twice[B])`. The types will match if the argument type of `twice[A]`, which is $A \Rightarrow A$, matches the type of `twice[B]`, which is $(B \Rightarrow B) \Rightarrow B \Rightarrow B$. Since the symbol $\Rightarrow$ groups to the right, we have

$$\begin{aligned} &(B \Rightarrow B) \Rightarrow B \Rightarrow B \\ &= (B \Rightarrow B) \Rightarrow (B \Rightarrow B) \quad . \end{aligned}$$

This can match with $A \Rightarrow A$ only if we set $A = (B \Rightarrow B)$. So the most general type of `twice(twice)` is

$$\text{twice}^{B \Rightarrow B}(\text{twice}^B) : (B \Rightarrow B) \Rightarrow B \Rightarrow B \quad . \tag{4.7}$$

After checking that types match, we may omit types from further calculations.

Example 4.3.1.2 defined `twice` with the `def` syntax. To use `twice` as an argument in the expression `twice(twice)`, it is convenient to define `twice` as a value,

```
val twice = ...
```

However, the function `twice` has type parameters, and Scala 2 does not directly support `val` definitions with type parameters. Scala 3 will support type parameters appearing together with value parameters in a nameless function:

```
val twice = [A] => (f: A => A) => (x: A) => f(f(x)) // Scala 3 only!
```

Keeping this in mind, we will use the curried definition of `twice` given by Eq. (4.6). Substituting this into the expression `twice(twice)`, we find

$$\begin{aligned} & \text{twice(twice)} \\ \text{definition of twice :} \quad & = (f \Rightarrow x \Rightarrow f(f(x)))(\text{twice}) \quad . \\ \text{substitute } f = \text{twice :} \quad & = x \Rightarrow \text{twice}(\text{twice}(x)) \\ \text{definition of twice :} \quad & = x \Rightarrow (f \Rightarrow x \Rightarrow f(f(x)))\,((f \Rightarrow x \Rightarrow f(f(x)))\,(x)) \quad . \end{aligned}$$

The last expression is hard to use: it is confusing that the argument names f and x are repeated. The calculation will be made clearer if we rename the arguments to remove shadowing of names. To avoid errors, we will start with $x \Rightarrow \text{twice}(\text{twice}(x))$ and rename arguments one scope at a time:

$$\begin{aligned}
& x \Rightarrow \text{twice}(\text{twice}(\underline{x})) \\
\text{rename } x \text{ to } z: \quad & = z \Rightarrow \underline{\text{twice}}(\text{twice}(z)) \\
\text{definition of twice}: \quad & = z \Rightarrow \underline{(f \Rightarrow x \Rightarrow f(f(x)))}\,(\text{twice}(z)) \\
\text{rename } f, x \text{ to } g, y: \quad & = z \Rightarrow (g \Rightarrow y \Rightarrow \underline{g}(\underline{g}(y)))\,(\text{twice}(z)) \\
\text{apply, } g = \text{twice}(z): \quad & = z \Rightarrow y \Rightarrow \underline{(\text{twice}(z))}\,(\text{twice}(z)(y)) \\
\text{use twice}(z) = (x \Rightarrow z(z(x))): \quad & = z \Rightarrow y \Rightarrow (x \Rightarrow z(z(x)))\,\underline{(\text{twice}(z)(y))} \\
\text{apply, } x = \text{twice}(z)(y) = z(z(y)): \quad & = z \Rightarrow y \Rightarrow z(z(z(z(y)))) \quad .
\end{aligned}$$

So `twice(twice)` is a function that applies its (function-typed) argument *four* times.

The type parameters follow from Eq. (4.7) with $A = \text{Int}$ and can be written as $\text{twice}^{\text{Int}\Rightarrow\text{Int}}(\text{twice}^{\text{Int}})$, or in Scala syntax, `twice[Int => Int](twice[Int])`. To test, we need to write at least one type parameter in the code, or else Scala cannot infer the types correctly:

```
scala> twice(twice[Int])(x => x + 3)(100) // _ + 3 + 3 + 3 + 3
res0: Int = 112

scala> twice[Int => Int](twice)(x => x + 3)(100)
res1: Int = 112
```

Example 4.3.1.4 Infer the type signature for the fully parametric function

```
def p[...]:... = { f => f(2) }
```

Can the types possibly match in the expression `p(p)`?

Solution In the nameless function $f \Rightarrow f(2)$, the argument f must be itself a function with an argument of type `Int`, otherwise the sub-expression $f(2)$ makes no sense. So, types will match if f has type $\text{Int} \Rightarrow \text{Int}$ or $\text{Int} \Rightarrow \text{String}$ or similar. The most general case is when f has type $\text{Int} \Rightarrow A$, where A is an arbitrary type (i.e. a type parameter). The type A will then be the (so far unknown) type of the value $f(2)$. Since nameless function $f \Rightarrow f(2)$ has an argument f of type $\text{Int} \Rightarrow A$ and the result of type A, we find that the type of p must be $(\text{Int} \Rightarrow A) \Rightarrow A$. With this type assignment, all types match. The type parameter A remains undetermined and is added to the type signature of the function `p`. The code is

```
def p[A]: (Int => A) => A = { f => f(2) }
```

To answer the question about the expression `p(p)`, we begin by writing that expression with new type parameters as `p[A](p[B])`. Then we try to choose `A` and `B` so that the types match in that expression. Does the type of `p[B]`, which is `(Int => B) => B`, match the type of the argument of `p[A]`, which is `Int => A`, with some choice of `A` and `B`? A function type `P => Q` matches `X => Y` only if `P = X` and `Q = Y`. So `(Int => B) => B` can match `Int => A` only if `Int => B` matches `Int` and if `B = A`. But it is impossible for `Int => B` to match `Int`, no matter how we choose `B`.

We conclude that types cannot be chosen consistently in `p[A](p[B])`. Such expressions contain a type error and are rejected by the Scala compiler. One also says that the expression `p(p)` is **not well-typed**, or does not **typecheck**.

For any given code expression containing only function expressions, one can always find the most general type that makes all functions match their arguments, unless the expression does not typecheck. The Damas-Hindley-Milner algorithm[2] performs type inference (or determines that there is

[2] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system#Algorithm_W

a type error) for a large class of expressions containing functions, tuples, and disjunctive types.

4.3.2 Calculations with curried functions

In mathematics, functions are evaluated by substituting their argument values into their body. Nameless functions are evaluated in the same way. For example, applying the nameless function $x \Rightarrow x + 10$ to an integer 2, we substitute 2 instead of x in "$x + 10$" and get "$2 + 10$", which we then evaluate to 12. The computation is written like this,

$$(x \Rightarrow x + 10)(2) = 2 + 10 = 12 \quad .$$

To run this computation in Scala, we need to add a type annotation:

```
scala> ((x: Int) => x + 10)(2)
res0: Int = 12
```

Curried function applications such as $f(x)(y)$ are rarely used in mathematics , so we need to gain some experience working with them.

Let us consider a curried nameless function being applied to arguments, such as `(x => y => x - y)(20)(4)`, and compute the result of this function application. Begin with the argument `20`; applying a nameless function of the form `(x => ...)` to `20` means to substitute `x = 20` into the body of the function. After that substitution, we obtain the expression `y => 20 - y`, which is again a nameless function. Applying that function to the remaining argument `(4)` means substituting `y = 4` into the body of that function. This yields the expression `20 - 4`. We can compute that and get the result, `16`. Check the result with Scala:

```
scala> ((x: Int) => (y: Int) => x - y)(20)(4)
res1: Int = 16
```

Applying a curried function such as `x => y => z => expr(x,y,z)` to three curried arguments `10`, `20`, and `30` means to substitute `x = 10`, `y = 20`, and `z = 30` into the expression `expr`. In this way, we can easily apply a curried function to any number of curried arguments.

This calculation is helped by the convention that `f(g)(h)` means first applying `f` to `g` and then applying the result to `h`. In other words, function application groups to the *left*: `f(g)(h) = (f(g))(h)`. It would be confusing if function application grouped to the right and `f(g)(h)` meant first applying `g` to `h` and then applying `f` to the result. If *that* were the syntax convention, it would be harder to reason about applying a curried function to the arguments.

We see that the right grouping of the function arrow `=>` is well adapted to the left grouping of function applications. All functional languages adopt these syntactic conventions.

To make calculations shorter, we will write code in a mathematical notation rather than in the Scala syntax. Type annotations are written with a colon in the superscript, for example: $x^{:\text{Int}} \Rightarrow x + 10$ instead of the code `((x:Int) => x + 10)`.

The symbolic evaluation of the Scala code `((x:Int) => (y:Int) => x - y)(20)(4)` can be performed as the following line-by-line derivation,

$$\begin{aligned} & (\underline{x^{:\text{Int}} \Rightarrow} \, y^{:\text{Int}} \Rightarrow \underline{x} - y)\,(20)\,(4) \\ \text{substitute } x = 20: \quad & = (\underline{y^{:\text{Int}} \Rightarrow} \, 20 - \underline{y})\,(4) \\ \text{substitute } y = 4: \quad & = 20 - 4 = 16 \quad . \end{aligned}$$

(The underlined part of the expression will be rewritten in the next line.)

Here we performed calculations by substituting an argument into a function at each step. A compiled Scala program is evaluated in a similar way at run time.

Nameless functions are *values* and so can be used as part of larger expressions, just as any other values. For instance, nameless functions can be arguments of other functions (nameless or not). Here is an example of applying a nameless function $f \Rightarrow f(2)$ to a nameless function $x \Rightarrow x + 4$:

$$\begin{aligned} & (f \Rightarrow \underline{f}(2))\,(x \Rightarrow x + 4) \\ \text{substitute } f = (x \Rightarrow x + 4): \quad & = (x \Rightarrow \underline{x} + 4)(2) \\ \text{substitute } x = 2: \quad & = 2 + 4 = 6 \quad . \end{aligned}$$

In the nameless function $f \Rightarrow f(2)$, the argument f has to be itself a function, otherwise the expression $f(2)$ would make no sense. The argument x of $f(x)$ must be an integer, or else we would not be able to compute $x + 4$. The result of computing $f(2)$ is 4, and integer. We conclude that in this example, f must have type Int ⇒ Int, or else the types will not match. To verify this result in Scala, we need to specify the type annotation for f:

```
scala> ((f: Int => Int) => f(2))(x => x + 4)
res2: Int = 6
```

No type annotation is needed for $x \Rightarrow x + 4$ since the Scala compiler already knows the type of f and can infer that x in $x \Rightarrow x + 4$ must have type `Int`.

To summarize the standard syntax conventions for curried nameless functions:

- Function expressions group everything to the right, so $x \Rightarrow y \Rightarrow z \Rightarrow e$ means $x \Rightarrow (y \Rightarrow (z \Rightarrow e))$.
- Function applications group everything to the left, so $f(x)(y)(z)$ means $((f(x))(y))(z)$.
- Function applications group stronger than infix operations, so $x + f(y)$ means $x + (f(y))$, as in mathematics.

Here are some more examples of performing function applications symbolically. Types are omitted for brevity; every non-function value is of type `Int`.

$$\begin{aligned} (x \Rightarrow x * 2)\,(10) &= 10 * 2 = 20 \quad . \\ (p \Rightarrow z \Rightarrow z * p)\,(t) &= (z \Rightarrow z * t) \quad . \\ (p \Rightarrow z \Rightarrow z * p)\,(t)(4) &= (z \Rightarrow z * t)(4) = 4 * t \quad . \end{aligned}$$

Some results of these computation are integer values such as 20; other results are nameless functions such as $z \Rightarrow z * t$. Verify this in Scala:

```
scala> ((x:Int) => x*2)(10)
res3: Int = 20

scala> ((p:Int) => (z:Int) => z*p)(10)
res4: Int => Int = <function1>

scala> ((p:Int) => (z:Int) => z*p)(10)(4)
res5: Int = 40
```

In the following examples, some arguments are themselves functions. We saw in Example 4.3.1.4 that the most general type for $g \Rightarrow g(2)$ is $(\text{Int} \Rightarrow A) \Rightarrow A$. Let us now compute an expression that uses $(g \Rightarrow g(2))$ as an argument:

$$\begin{aligned} & (f \Rightarrow p \Rightarrow \underline{f}(p))\,(g \Rightarrow g(2)) \\ \text{substitute } f = (g \Rightarrow g(2)): \quad & = p \Rightarrow (g \Rightarrow \underline{g}(2))\,(p) \\ \text{substitute } g = p: \quad & = p \Rightarrow p(2) \quad . \end{aligned} \tag{4.8}$$

The result of this expression is a function $p \Rightarrow p(2)$ that will apply *its* argument to the value 2. A possible argument is the function $x \Rightarrow x + 4$. So, let us apply the previous expression to that function:

$$\begin{aligned} & \underline{(f \Rightarrow p \Rightarrow f(p))\,(g \Rightarrow g(2))}\,(x \Rightarrow x + 4) \\ \text{use Eq. (4.8)}: \quad & = (p \Rightarrow \underline{p}(2))\,(x \Rightarrow x + 4) \\ \text{substitute } p = (x \Rightarrow x + 4): \quad & = (x \Rightarrow \underline{x} + 4)\,(2) \\ \text{substitute } x = 2: \quad & = 2 + 4 = 6 \quad . \end{aligned}$$

To verify this calculation in Scala, we need to add appropriate type annotations for f and p. To figure out the types, we reason like this:

We know that the function $f \Rightarrow p \Rightarrow f(p)$ is being applied to the arguments $f = g \Rightarrow g(2)$ and $p = x \Rightarrow x + 4$. The most general type of $g \Rightarrow g(2)$ is $(\text{Int} \Rightarrow A) \Rightarrow A$. So, this must be the type of the argument f in $f \Rightarrow p \Rightarrow f(p)$.

The variable x in $x \Rightarrow x + 4$ must be of type `Int`, or else we cannot add x to 4. Thus, the type of the expression $x \Rightarrow x + 4$ is $\text{Int} \Rightarrow \text{Int}$, and so must be the type of the argument p in $f \Rightarrow p \Rightarrow f(p)$.

Finally, we need to make sure that the types match in the function $f \Rightarrow p \Rightarrow f(p)$. The types match in $f(p)$ if the type of f's argument is the same as the type of p. Since f has type $(\text{Int} \Rightarrow A) \Rightarrow A$, its argument has type $\text{Int} \Rightarrow A$. To match that with the type of $p^{:\text{Int} \Rightarrow \text{Int}}$ requires us to set $A = \text{Int}$. So, the actual type of f is $(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$. We know enough to write the Scala code now:

```
scala> ((f: (Int => Int) => Int) => p => f(p))(g => g(2))(x => x + 4)
res6: Int = 6
```

Type annotations for p, g, and x may be omitted because the Scala compiler will infer the missing types unambiguously from the given type of f. However, it is never an error to specify more types; it just makes the code longer.

4.4 Summary

What can we do with this chapter's techniques?

- Implement functions that return new functions and/or take functions as arguments.
- Simplify function applications symbolically.
- Infer the most general type for a given code expression.
- Convert functions to a fully parametric form.

The following solved examples and exercises illustrate these techniques.

4.4.1 Solved examples

Example 4.4.1.1 Implement a function that applies a given function f repeatedly to an initial value x_0, until a given condition function `cond` returns `true`:

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X = ???
```

Solution We create an iterator that keeps applying the function f, and use `.find` to stop the sequence when the condition first holds:

```
def converge[X](f: X => X, x0: X, cond: X => Boolean): X =
  Stream.iterate(x0)(f) // Type is Stream[X].
  .find(cond) // Type is Option[X].
  .get // Type is X.
```

Here it is safe to perform `.get` on an `Option`: If the condition never holds, the program will run out of memory (since `Stream.iterate` keeps all computed values in memory) or the user runs out of time.

A tail-recursive implementation that works in constant memory is

```
@tailrec def converge[X](f: X => X, x0: X, cond: X => Boolean): X =
  if (cond(x0)) x0 else converge(f, f(x0), cond)
```

To test this code, let us compute an approximation to the square root of a number q by Newton's method. The iteration function f is

$$f(x) = \frac{1}{2}\left(x + \frac{q}{x}\right) \quad .$$

We iterate $f(x)$ starting with $x_0 = q/2$ until we obtain a given precision:

```
def approx_sqrt(x: Double, precision: Double): Double = {
      def cond(y: Double): Boolean = math.abs(y * y - x) <= precision
      def iterate_sqrt(y: Double): Double = 0.5 * (y + x / y)
      converge(iterate_sqrt, x / 2, cond)
}
```

Test it:

```
scala> approx_sqrt(25, 1.0e-8)
res0: Double = 5.000000000016778
```

Example 4.4.1.2 Using both `def` and `val`, define a Scala function that takes an integer `x` and returns a function that adds `x` to *its* argument.

Solution Let us first write down the required type signature: the function must take an integer argument `x:Int`, and the return value must be a function of type `Int => Int`.

```
def add_x(x: Int): Int => Int = ???
```

We are required to return a function that adds `x` to its argument. Let us call that argument `z`, to avoid confusion with the `x`. So, we are required to return a function that we can write as `{ z => z + x }`. Since functions are values, we can directly return a new function by writing a nameless function expression:

```
def add_x(x: Int): Int => Int = { z => z + x }
```

To implement the same function via a `val`, we first convert the type signature of `add_x` to the equivalent type expression Int ⇒ Int ⇒ Int. Now we can write the Scala code of a function `add_x_v`:

```
val add_x_v: Int => Int => Int = { x => z => z + x }
```

The function `add_x_v` is equal to `add_x` except for using the `val` syntax instead of `def`. It is not necessary to specify the type of the arguments `x` and `z` because we already specified the type Int ⇒ Int ⇒ Int for the value `add_x_v`.

Example 4.4.1.3 Using both `def` and `val`, implement a curried function `prime_f` that takes a function f and an integer x, and returns `true` when $f(x)$ is a prime number. Use the function `is_prime` defined in Section 1.1.2.

Solution First, we need to determine the required type signature of `prime_f`. The value $f(x)$ must type `Int`, or else we cannot check whether it is prime. So, f must have type Int ⇒ Int. Since `prime_f` should be a curried function, we need to put each argument into its own set of parentheses:

```
def prime_f(f: Int => Int)(x: Int): Boolean = ???
```

To implement `prime_f`, we need to return the result of `is_prime` applied to `f(x)`. A simple solution is

```
def prime_f(f: Int => Int)(x: Int): Boolean = is_prime(f(x))
```

To implement the same function as a `val`, rewrite its type signature as

```
val prime_f: (Int => Int) => Int => Boolean = ???
```

(The parentheses around `Int => Int` are mandatory since `Int => Int => Int` would be a completely different type.) The implementation is

```
val prime_f: (Int => Int) => Int => Boolean = { f => x => is_prime(f(x)) }
```

We also notice that the code is a forward composition of the functions `f` and `is_prime`, so we can write it as

```
val prime_f: (Int => Int) => Int => Boolean = (f => f andThen is_prime)
```

The function body is of the form `f => f.something`, which is equivalent to a shorter Scala syntax `(_.something)`. So we can rewrite the code of `prime_f` as

```
val prime_f: (Int => Int) => Int => Boolean = (_ andThen is_prime)
```

Example 4.4.1.4 Implement a function `choice(x,p,f,g)` that takes a value x, a predicate p, and two functions f and g. The return value must be $f(x)$ if $p(x)$ returns true; otherwise the return value must be $g(x)$. Infer the most general type for this function.

Solution The code of this function must be

```
def choice[...](x,p,f,g) = if (p(x)) f(x) else g(x)
```

Now let us infer the most general type for this code. We begin by assuming that x has type A, where A is a type parameter. Then the predicate p must have type `A => Boolean`. Since p is an arbitrary predicate, the value $p(x)$ will be sometimes `true` and sometimes `false`. So, `choice(x,p,f,g)` will sometimes return $f(x)$ and sometimes $g(x)$. It follows that type A must be the argument type of both f and g, which means that the most general types so far are $f^{:A\Rightarrow B}$ and $g^{:A\Rightarrow C}$, yielding the type signature

$$\text{choice}(x^{:A}, p^{:A\Rightarrow\text{Boolean}}, f^{:A\Rightarrow B}, g^{:A\Rightarrow C}) \quad .$$

What could be the return type of `choice(x,p,f,g)`? If $p(x)$ returns `true`, the function `choice` returns $f(x)$, which is of type B. Otherwise, `choice` returns $g(x)$, which is of type C. However, the type signature of `choice` must be fixed in advance (at compile time) and cannot depend on the value $p(x)$ computed at run time. So, the types of $f(x)$ and of $g(x)$ must be the same, $B = C$. The type signature of `choice` will thus have only two type parameters, A and B:

```
def choice[A, B](x: A, p: A => Boolean, f: A => B, g: A => B): B =
  if (p(x)) f(x) else g(x)
```

Example 4.4.1.5 Infer the most general type for the fully parametric function

```
def q[...]: ... = { f => g => g(f) }
```

What types are inferred for the expressions `q(q)` and `q(q(q))`?

Solution Begin by assuming $f^{:A}$ for an arbitrary type A. In the sub-expression $g \Rightarrow g(f)$, the curried argument g must itself be a function, because it is being applied to f as $g(f)$. So we assign types as $f^{:A} \Rightarrow g^{:A\Rightarrow B} \Rightarrow g(f)$, where A and B are some type parameters. Since there are no other constraints on the types, the parameters A and B remain arbitrary, so we add them to the type signature:

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

To match types in the expression `q(q)`, we first assume arbitrary type parameters and write `q[A, B](q[C, D])`. We need to introduce new type parameters C, D because these type parameters will probably need to be set differently from A, B when we try to match the types in the expression `q(q)`.

The type of the first curried argument of `q[A, B]`, which is A, must match the entire type of `q[C, D]`, which is $C \Rightarrow (C \Rightarrow D) \Rightarrow D$. So we must set the type parameter A as

$$A = C \Rightarrow (C \Rightarrow D) \Rightarrow D \quad .$$

The type of `q(q)` becomes

$$q^{A,B}(q^{C,D}) : ((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B \quad ,$$
$$\text{where} \quad A = C \Rightarrow (C \Rightarrow D) \Rightarrow D \quad .$$

We use this result to infer the most general type for `q(q(q))`. We may denote $r \triangleq q(q)$ for brevity; then, as we just found, r has type $((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B$. To infer types in the expression `q(r)`, we introduce new type parameters E, F and write `q[E, F](r)`. The type of the argument of `q[E, F]` is E, and this must be the same as the type of r. This gives the constraint

$$E = ((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B \quad .$$

Other than that, the type parameters are arbitrary. The type of the expression `q(q(q))` is $(E \Rightarrow F) \Rightarrow F$. We conclude that the most general type of `q(q(q))` is

$$q^{E,F}(q^{A,B}(q^{C,D})) : ((((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B) \Rightarrow F) \Rightarrow F \quad ,$$
$$\text{where} \quad A = C \Rightarrow (C \Rightarrow D) \Rightarrow D$$
$$\text{and} \quad E = ((C \Rightarrow (C \Rightarrow D) \Rightarrow D) \Rightarrow B) \Rightarrow B \quad .$$

It is clear from this derivation that all expressions such as `q(q(q(q)))`, `q(q(q(q(q))))`, etc., are well-typed.

Let us test these results in Scala, renaming the type parameters for clarity to `A`, `B`, `C`, `D`:

```
scala> def qq[A, B, C]: ((A => (A => B) => B) => C) => C = q(q)
qq: [A, B, C]=> ((A => ((A => B) => B)) => C) => C

scala> def qqq[A, B, C, D]: ((((A => (A => B) => B) => C) => C) => D) => D = q(q(q))
qqq: [A, B, C, D]=> ((((A => ((A => B) => B)) => C) => C) => D) => D
```

We did not need to write any type parameters within the expressions `q(q)` and `q(q(q))` because the full type signature was declared at the beginning of each of these expressions. Since the Scala compiler did not print any error messages, we are assured that the types match correctly.

Example 4.4.1.6 Infer types in the code expression

$$(f \Rightarrow g \Rightarrow g(f))\,(f \Rightarrow g \Rightarrow g(f))\,(f \Rightarrow f(10)) \quad ,$$

and simplify the code.

Solution The given expression is a curried function $f \Rightarrow g \Rightarrow g(f)$ applied to two curried arguments. The plan is to consider each of these sub-expressions in turn, assigning types for them using type parameters, and then to figure out how to set the type parameters so that all types match.

Begin by renaming the shadowed variables f and g, to remove shadowing:

$$(f \Rightarrow g \Rightarrow g(f))\,(x \Rightarrow y \Rightarrow y(x))\,(h \Rightarrow h(10)) \quad . \tag{4.9}$$

As we have seen in Example 4.4.1.5, the sub-expression $f \Rightarrow g \Rightarrow g(f)$ is typed as $f^{:A} \Rightarrow g^{:A \Rightarrow B} \Rightarrow g(f)$, where A and B are some type parameters. The sub-expression $x \Rightarrow y \Rightarrow y(x)$ is the same function as $f \Rightarrow g \Rightarrow g(f)$ but with possibly different type parameters, say, $x^{:C} \Rightarrow y^{:C \Rightarrow D} \Rightarrow y(x)$. The types A, B, C, D are so far unknown.

Finally, the variable h in the sub-expression $h \Rightarrow h(10)$ must have type $\text{Int} \Rightarrow E$, where E is another type parameter. So, the sub-expression $h \Rightarrow h(10)$ is a function of type $(\text{Int} \Rightarrow E) \Rightarrow E$.

The types must match in the entire expression (4.9):

$$(f^{:A} \Rightarrow g^{:A \Rightarrow B} \Rightarrow g(f))(x^{:C} \Rightarrow y^{:C \Rightarrow D} \Rightarrow y(x))(h^{:\text{Int} \Rightarrow E} \Rightarrow h(10)) \quad . \tag{4.10}$$

It follows that f must have the same type as $x \Rightarrow y \Rightarrow y(x)$, and g must have the same type as $h \Rightarrow h(10)$. The type of g, which we know as $A \Rightarrow B$, will match the type of $h \Rightarrow h(10)$, which we know as $(\text{Int} \Rightarrow E) \Rightarrow E$, only if $A = \text{Int} \Rightarrow E$ and $B = E$. It follows that f has type $\text{Int} \Rightarrow E$. At the same time, the type of f must match the type of $x \Rightarrow y \Rightarrow y(x)$, which is $C \Rightarrow (C \Rightarrow D) \Rightarrow D$. This can work only if $C = \text{Int}$ and $E = (C \Rightarrow D) \Rightarrow D$.

In this way, we have found all the relationships between the type parameters A, B, C, D, E in Eq. (4.10). The type D remains undetermined (i.e. arbitrary), while the type parameters A, B, C, E are expressed as

$$A = \text{Int} \Rightarrow (\text{Int} \Rightarrow D) \Rightarrow D \quad , \tag{4.11}$$

$$B = E = (\text{Int} \Rightarrow D) \Rightarrow D \quad , \tag{4.12}$$

$$C = \text{Int} \quad .$$

The entire expression in Eq. (4.10) is a saturated application of a curried function, and thus has the same type as the "final" result expression $g(f)$, which has type B. So, the entire expression in Eq. (4.10) has type $B = (\text{Int} \Rightarrow D) \Rightarrow D$.

Having established that types match, we can now omit the type annotations and rewrite the code expression as

$$\begin{aligned} & (f \Rightarrow g \Rightarrow \underline{g}(\underline{f}))\,(x \Rightarrow y \Rightarrow y(x))\,(h \Rightarrow h(10)) \\ \text{substitute } f, g: \quad & = (h \Rightarrow \underline{h}(10))\,(x \Rightarrow y \Rightarrow y(x)) \\ \text{substitute } h: \quad & = (x \Rightarrow y \Rightarrow y(\underline{x}))(10) \\ \text{substitute } x: \quad & = y \Rightarrow y(10) \quad . \end{aligned}$$

The type of this expression is $(\text{Int} \Rightarrow D) \Rightarrow D$ with a type parameter D. Since the argument y is an arbitrary function, we cannot simplify $y(10)$ or $y \Rightarrow y(10)$ any further. We conclude that $y^{:\text{Int} \Rightarrow D} \Rightarrow y(10)$ is the final simplified form of Eq. (4.9).

To test this, we first define the function $f \Rightarrow g \Rightarrow g(f)$ as in Example 4.4.1.5,

```
def q[A, B]: A => (A => B) => B = { f => g => g(f) }
```

We also define the function $h \Rightarrow h(10)$ with a general type $(\text{Int} \Rightarrow E) \Rightarrow E$,

```
def r[E]: (Int => E) => E = { h => h(10) }
```

To help Scala evaluate Eq. (4.10), we need to set the type parameters for the first `q` function as `q[A, B]` where A and B are given by Eqs. (4.11)–(4.12):

```
scala> def s[D] = q[Int => (Int => D) => D, (Int => D) => D](q)(r)
s: [D]=> (Int => D) => D
```

To verify that the function s^D indeed equals $y^{:\text{Int} \Rightarrow D} \Rightarrow y(10)$, we apply s^D to some functions of type $\text{Int} \Rightarrow D$, say, for $D = \text{Boolean}$ or $D = \text{Int}$:

```
scala> s(_ > 0) // Evaluate 10 > 0.
res6: Boolean = true

scala> s(_ + 20) // Evaluate 10 + 20.
res7: Int = 30
```

4.4.2 Exercises

Exercise 4.4.2.1 For `id` and `const` as defined above, what are the types of `id(id)`, `id(id)(id)`, `id(id(id))`, `id(const)`, and `const(const)`? Simplify these expressions.

Exercise 4.4.2.2 For the function `twice` from Example 4.3.1.2, infer the most general type for `twice(twice(twice))`. What does that function do? Test your answer.

Exercise 4.4.2.3 Define a function `thrice` similarly to `twice` except it should apply a given function 3 times. What does the function `thrice(thrice(thrice))` do?

Exercise 4.4.2.4 Define a function `ence` similarly to `twice` except it should apply a given function n times, where n is an additional curried argument.

Exercise 4.4.2.5 Define a fully parametric function `flip(f)` that swaps arguments for any given function `f` of two arguments. To test:

```
def f(x: Int, y: Int) = x - y // Check that f(10, 2) == 8.
val g = flip(f)  // Now check that g(2, 10) == 8.
```

Exercise 4.4.2.6 Revise the function from Exercise 1.6.2.4, implementing it a curried function and replacing the hard-coded number 100 by a *curried* first argument. The type signature should become `Int => List[List[Int]] => List[List[Int]]`.

Exercise 4.4.2.7 Implement the function `converge` from Example 4.4.1.1 as a curried function, with an additional argument to set the maximum number of iterations, and returning `Option[Double]` as the final result type. The new version of `converge` should return `None` if the convergence condition is not satisfied after the given maximum number of iterations. The type signature and an example test:

```
@tailrec def convergeN[X](p:X => Boolean)(x:X)(m:Int)(f:X => X): Option[X] = ???

scala> convergeN[Int](_ < 0)(0)(10)(_ + 1)
res0: Option[Int] = None

scala> convergeN[Double]{ x => math.abs(x * x - 25) < 1e-8 }(1.0)(10) { x => 0.5 * (x + 25 / x ) }
res1: Option[Double] = Some(5.000000000053722)
```

Exercise 4.4.2.8 Write a function `curry2` converting an uncurried function of type `(Int, Int) => Int` into an equivalent curried function of type `Int => Int => Int`.

Exercise 4.4.2.9 Apply the function $(x \Rightarrow \_ \Rightarrow x)$ to the value $(z \Rightarrow z(q))$ where q is a given value of type Q. Infer types in these expressions.

Exercise 4.4.2.10 Infer types in the following expressions:

$$\text{(a):}\quad p \Rightarrow q \Rightarrow p(t \Rightarrow t(q)) \quad ,$$
$$\text{(b):}\quad p \Rightarrow q \Rightarrow q(x \Rightarrow x(p(q))) \quad .$$

Exercise 4.4.2.11 Show that the following expressions cannot be well-typed:

$$\text{(a):}\quad p \Rightarrow p(q \Rightarrow q(p)) \quad ,$$
$$\text{(b):}\quad p \Rightarrow q \Rightarrow q(x \Rightarrow p(q(x))) \quad .$$

Exercise 4.4.2.12 Infer types and simplify the following code expressions:

$$\begin{aligned}
&\text{(a):} \quad q \Rightarrow (x \Rightarrow y \Rightarrow z \Rightarrow x(z)(y(z)))\,(a \Rightarrow a)\,(b \Rightarrow b(q)) \quad , \\
&\text{(b):} \quad (f \Rightarrow g \Rightarrow h \Rightarrow f(g(h)))\,(x \Rightarrow x) \quad , \\
&\text{(c):} \quad (x \Rightarrow y \Rightarrow x(y))\,(x \Rightarrow y \Rightarrow x) \quad , \\
&\text{(d):} \quad (x \Rightarrow y \Rightarrow x(y))\,(x \Rightarrow y \Rightarrow y) \quad , \\
&\text{(e):} \quad x \Rightarrow (f \Rightarrow y \Rightarrow f(y)(x))\,(z \Rightarrow _ \Rightarrow z) \quad , \\
&\text{(f):} \quad z \Rightarrow (x \Rightarrow y \Rightarrow x)\,(x \Rightarrow x(z))\,(y \Rightarrow y(z)) \quad .
\end{aligned}$$

4.5 Discussion

4.5.1 Higher-order functions

The **order** of a function is the number of function arrows (`=>`) contained in the type signature of that function. If a function's type signature contains more than one arrow, the function is called a **higher-order** function. A higher-order function takes a function as argument and/or returns a function as its result value.

The methods `andThen`, `compose`, `curried`, and `uncurried` are examples of higher-order functions that take other functions as arguments *and* return a new function.

The following example illustrates the concept of a function's order. Consider

```
def f1(x: Int): Int = x + 10
```

The function `f1` has type signature `Int => Int` and order 1, so it is *not* a higher-order function.

```
def f2(x: Int): Int => Int = z => z + x
```

The function `f2` has type signature `Int => Int => Int` and is a higher-order function of order 2.

```
def f3(g: Int => Int): Int = g(123)
```

The function `f3` has type signature `(Int => Int) => Int` and is a higher-order function of order 2.

Although `f2` is a higher-order function, its "higher-orderness" comes from the fact that the return value is of a function type. An equivalent computation can be performed by an uncurried function that is not higher-order:

```
scala> def f2u(x: Int, z: Int): Int = z + x
```

Unlike `f2`, the function `f3` *cannot* be converted to a non-higher-order function because `f3` has an argument of a function type. Converting to an uncurried form cannot eliminate an argument of a function type.

4.5.2 Name shadowing and the scope of bound variables

Bound variables are introduced in nameless functions whenever an argument is defined. For example, in the curried nameless function $x \Rightarrow y \Rightarrow f(x, y)$, the bound variables are the curried arguments x and y. The variable y is only defined within the scope $(y \Rightarrow f(x, y))$ of the inner function; the variable x is defined within the entire scope of $x \Rightarrow y \Rightarrow f(x, y)$.

Another way of introducing bound variables in Scala is to write a `val` or a `def` within curly braces:

```
val x = {
  val y = 10 // Bound variable.
  y + y * y
} // Same as ‘val x = 10 + 10 * 10‘.
```

A bound variable is invisible outside the scope that defines it. This is why bound variables may be renamed at will: no outside code could possibly use them and depend on their values. However, outside code may define a variable that (by chance) has the same name as a bound variable inside the scope.

Consider this example from calculus: In the integral

$$f(x) = \int_0^x \frac{dx}{1+x} \quad ,$$

a bound variable named x is defined in *two* local scopes: in the scope of f and in the scope of the nameless function $x \Rightarrow \frac{1}{1+x}$. The convention in mathematics is to treat these two x's as two *completely different* variables that just happen to have the same name. In sub-expressions where both of these bound variables are visible, priority is given to the bound variable defined in the closest inner scope. The outer definition of x is **shadowed**, i.e. hidden, by the definition of the inner x. For this reason, mathematicians expect that evaluating $f(10)$ will give

$$f(10) = \int_0^{10} \frac{dx}{1+x} \quad ,$$

rather than $\int_0^{10} \frac{dx}{1+10}$, because the outer definition $x = 10$ is shadowed within the expression $\frac{1}{1+x}$ by the closer definition of x in the local scope of $x \Rightarrow \frac{1}{1+x}$.

Since this is the standard mathematical convention, the same convention is adopted in functional programming. A variable defined in a function scope (i.e. a bound variable) is invisible outside that scope but will shadow any outside definitions of a variable with the same name.

Name shadowing is not advisable, because it usually decreases the clarity of code and so invites errors. Consider the function

$$x \Rightarrow x \Rightarrow x \quad .$$

Let us decipher this confusing syntax. The symbol $\Rightarrow$ groups to the right, so $x \Rightarrow x \Rightarrow x$ is the same as $x \Rightarrow (x \Rightarrow x)$. It is a function that takes x and returns $x \Rightarrow x$. Since the nameless function $(x \Rightarrow x)$ may be renamed to $(y \Rightarrow y)$ without changing its value, we can rewrite the code to

$$x \Rightarrow (y \Rightarrow y) \quad .$$

Having removed name shadowing, we can more easily understand this code and reason about it. For instance, it becomes clear that this function ignores its argument x and always returns the same value (the identity function $y \Rightarrow y$).

4.5.3 Operator syntax for function applications

In mathematics, function applications are sometimes written without parentheses, for instance $\cos x$ or $\sin z$. There are also cases where formulas such as $2 \sin x \cos x$ imply parentheses as $2 \cdot \sin(x) \cdot \cos(x)$. Functions such as $\cos x$ are viewed as "operators" that are applied to their arguments without parentheses, similar to the operators of summation, $\sum_k k$, and differentiation, $\partial_x f$.

Many programming languages (such as ML, OCaml, F#, Haskell, Elm, PureScript) have adopted this "operator syntax", making parentheses optional for function arguments. The result is a syntax where `f x` means the same as `f(x)`. Parentheses are still used where necessary to avoid ambiguity or for readability.[3]

The conventions for nameless functions in the operator syntax become:

[3] The operator syntax has a long history in programming. It is used in Unix shell commands, for example `cp file1 file2`. In LISP and Scheme, function applications are enclosed in parentheses but the arguments are separated by spaces, for example `(f 10)`.

- Function expressions group everything to the right, so $x \Rightarrow y \Rightarrow z \Rightarrow e$ means $x \Rightarrow (y \Rightarrow (z \Rightarrow e))$.
- Function applications group everything to the left, so $f\ x\ y\ z$ means $((f\ x)y)z$.
- Function applications group stronger than infix operations, so $x + f\ y$ means $x + (f\ y)$, just as in mathematics $x + \cos y$ groups $\cos y$ stronger than the infix "+" operation.

Thus, $x \Rightarrow y \Rightarrow a\,b\,c + p\,q$ means $x \Rightarrow (y \Rightarrow ((a\,b)\ c) + (p\,q))$. When this notation becomes hard to read correctly, one needs to add parentheses, e.g. to write $f(x \Rightarrow g\ h)$ instead of $f\ x \Rightarrow g\ h$.

This book will avoid using the "operator syntax" when reasoning about code. Scala does not support the parentheses-free operator syntax; parentheses need to be put around every curried argument.

From the point of view of programming language theory, curried functions are "simpler" because they always have a *single* argument (and may return a function that will consume further arguments). From the point of view of programming practice, curried functions are often harder to read and to write.

In the operator syntax used e.g. in OCaml and Haskell, a curried function `f` is applied to curried arguments as, e.g., `f 20 4`. This departs further from the mathematical tradition and requires some getting used to. If the two arguments are more complicated than just 20 and 4, the resulting expression may become harder to read, compared with the syntax where commas are used to separate the arguments. (Consider e.g. the expression `f (g 10) (h 20) + 30`.) To improve readability of code, programmers may prefer to first define short names for complicated expressions and then use these names as curried arguments.

In Scala, the choice of whether to use curried or uncurried function signatures is largely a matter of syntactic convenience. Most Scala code tends to be written with uncurried functions, while curried functions are used when they produce more easily readable code.

One of the syntactic features for curried functions in Scala is the ability to specify a curried argument using the curly brace syntax. Compare the two definitions of the function `summation` described in Section 1.7.5:

```
def summation1(a: Int, b: Int, g: Int => Int): Int =
  (a to b).map(g).sum

def summation2(a: Int, b: Int)(g: Int => Int): Int =
  (a to b).map(g).sum
```

These functions are used in the code with a slightly different syntax:

```
scala> summation1(1, 10, { x => x*x*x + 2*x })
res0: Int = 3135

scala> summation2(1, 10){ x => x*x*x + 2*x }
res1: Int = 3135
```

The code that calls `summation2` is easier to read because the curried argument is syntactically separated from the rest of the code by curly braces. This is especially useful when the curried argument is itself a function with a complicated body, since Scala's curly braces syntax allows function bodies to contain their own local definitions (`val` or `def`).

Another feature of Scala is the "dotless" method syntax: for example, `xs map f` is equivalent to `xs.map(f)` and `f andThen g` is equivalent to `f.andThen(g)`. The "dotless" syntax is available only for infix methods, such as `.map`, defined on specific types such as `Seq`. In Scala 3, the "dotless" syntax will only work for methods having a special `@infix` annotation. Do not confuse Scala's "dotless" method syntax with the operator syntax used in Haskell and some other languages.

4.5.4 Deriving a function's code from its type signature

We have seen how the procedure of type inference derives the type of a fully parametric function from the function's code. It is remarkable that one can sometimes derive the function's *code* from the function's type signature. We will now look at some examples of this.

Consider a fully parametric function that performs a partial application for arbitrary other functions. A possible type signature is

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = ???
```

The function `pa` will substitute a fixed argument value `x:A` into another given function `f`.

How can we implement `pa`? Since `pa(x)(f)` must return a function of type `B => C`, we have no choice other than to begin writing the function body as

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = { y: B =>
  ??? // Need to compute a value of type C in this scope.
}
```

In the inner scope, we need to compute a value of type `C`, and we have values `x:A`, `y:B`, and `f:(A, B)=>C`. How can we compute a value of type `C`? If we knew that `C = Int` when `pa(x)(f)` is applied, we could have simply selected a fixed integer value, say, `1`, as the value of type `C`. If we knew that `C = String`, we could have selected a fixed string, say, `"hello"`, as the value of type `C`. But a fully parametric function cannot use any knowledge of the actual types used in the code when that function is being applied to arguments.

So, a fully parametric function cannot produce a value of an arbitrary type `C` from scratch. The only way of producing a value of type `C` is by applying the function `f` to arguments of types `A` and `B`. Since the types `A` and `B` are arbitrary, we cannot obtain any values of these types other than `x:A` and `y:B`. So, the only way of obtaining a value of type `C` is to compute `f(x, y)`. Thus, the body of `pa` must be

```
def pa[A, B, C](x: A)(f: (A, B) => C): B => C = { y => f(x, y) }
```

In this way, we have *unambiguously* derived the body of this function from its type signature, by assuming that the function must be fully parametric.

Another example is the operation of forward composition $f \fatsemi g$, viewed as a fully parametric function with type signature

```
def before[A, B, C](f: A => B, g: B => C): A => C = ???
```

To implement `before`, we need to create a nameless function of type `A => C`,

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x: A =>
  ??? // Need to compute a value of type C in this scope.
}
```

In the inner scope, we need to compute a value of type C, and we have the values $x^{:A}$, $f^{:A \Rightarrow B}$, and $g^{:B \Rightarrow C}$. Since the type C is arbitrary, the only way of obtaining a value of type C is to apply g to an argument of type B. The only way of obtaining a value of type B is to apply f to an argument of type A. Finally, we have only one value of type A, namely $x^{:A}$. So, the only way of obtaining the required result is to compute $g(f(x))$. We have again unambiguously derived the body of the function before from its type signature:

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x => g(f(x)) }
```

In Chapter 5 and in Appendix **??**, we will see how a function's code can be derived from type signatures for a wide range of fully parametric functions.

5 The logic of types. III. The Curry-Howard correspondence

Fully parametric functions were defined in Section 4.2. These functions perform general operations that work with type parameters and do not depend on any specific data types. We have seen examples of fully parametric functions, such as

```
def before[A, B, C](f: A => B, g: B => C): A => C = { x => g(f(x)) }
```

We have also seen in Section 4.5.4 that for certain functions of this kind, the code can be derived unambiguously from the type signature.

There exists a mathematical theory (called the **Curry-Howard correspondence**) that provides precise conditions for the possibility of deriving a function's code from its type, and, when possible, a systematic derivation algorithm. Technical details about the algorithm are found in Appendix **??**. This chapter will describe the main results and applications of this theory to functional programming.

5.1 Values computed by fully parametric functions

5.1.1 Motivation

To begin, consider the Scala code of a fully parametric function,

```
def f[A, B, ...]: ... = {
  ...
  val x: A = ... // Some expression here.
  ...
}
```

If this program compiles without type errors, it means that the types match and, in particular, that the function `f` is able to compute a value `x` of type `A`.

It is sometimes *impossible* to compute a value of a certain type within the body of a fully parametric function. For example, the fully parametric function `fmap` described in Section 3.2.3.1 cannot compute any values of type `A`,

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  val x: A = ??? // Cannot compute x here!
  ... }
```

The reason is that a fully parametric function cannot compute values of type `A` from scratch, without using previously given values of type `A` and without applying a function that returns values of type `A`. In `fmap`, no values of type `A` are given as arguments; the given function `f: A => B` returns values of type `B` and not `A`. The code of `fmap` must do pattern matching on `oa:Option[A]`, yielding two cases:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None       =>
    val x: A = ??? // Cannot compute x here!
  case Some(a)    =>
    val x: A = a   // Can compute x in this scope.
}
```

Since the case `None` does not contain any values of type `A`, we are unable to compute a value `x` in that scope (as long as `fmap` remains a fully parametric function). Being able to compute `x:A` "within the body of the function" means that, if needed, function should be able to return `x` as a result value. This requires being able to compute `x` in *all* cases, not just within one part of the `match` expression.

The body of `fmap` also cannot compute any values of type `B`. Since no arguments of type `B` are given, the only way of obtaining a value of type `B` would be to apply the function `f: A => B` to *some* value of type `A`; but we just saw that the body of `fmap` cannot compute any values of type `A`.

Another example of being unable to compute a value of a certain type is

```
def before[A, B, C](f: A => B, g: B => C): A => C = {
  //  val h: C => A = ??? // Cannot compute h here!
  a => g(f(a)) // Can compute a value of type A => C.
}
```

The body of `before` may only use the arguments `f` and `g`. It is possible to obtain a value of type `A => C` by composing `f` and `g`, but it is impossible to compute a value `h` of type `C => A`, no matter what code we try to write for computing `h` via `f` and `g`. The reason is that the body of `before` has no given values of type `A` and no functions that return values of type `A`, so a nameless function such as `(c:C) => ???` cannot implement its return value of type `A`. Since a fully parametric function cannot create values of an arbitrary type `A` from scratch, we see no possibility of computing `h` within the body of `before`.

Can we prove rigorously that a value of type `C => A` cannot be computed within the body of `before`? Could some clever trick produce a value of that type? So far, we only gave informal arguments about whether values of certain types can be computed. To make the arguments rigorous, we need to translate statements such as "*a fully parametric function* `before` *can compute a value of type* `C => A`" into mathematical formulas having rigorous rules for proving them true or false.

In Section 3.5.3, we denoted by $\mathcal{CH}(A)$ the proposition "the $\mathcal{C}$ode $\mathcal{H}$as a value of type A". By "the code" we now mean the body of a given fully parametric function. So, the notation $\mathcal{CH}(A)$ is not fully adequate because the validity of the proposition $\mathcal{CH}(A)$ depends not only on the choice of the type A but also on the function in which the value of type A needs to be computed. What exactly is this additional dependency? In our reasoning in the above examples, we used the types of a function's *arguments* in order to analyze the possibility of computing a value of a given type A. Thus, a precise description of the proposition $\mathcal{CH}(A)$ is

$$\begin{aligned} &\mathcal{CH}\text{-proposition}: \quad \text{a fully parametric function having arguments of types} \\ &\qquad\qquad X, Y, ..., Z \text{ can compute a value of type } A \quad . \end{aligned} \tag{5.1}$$

Here, $X, Y, ..., Z, A$ may be type parameters or more complicated type expressions such as $X = B \Rightarrow C$ or $Y = (C \Rightarrow D) \Rightarrow E$, built from other type parameters.

If arguments of types X, Y, ..., Z are already given, the propositions $\mathcal{CH}(X)$, $\mathcal{CH}(Y)$, ..., $\mathcal{CH}(Z)$ will be true. So, proposition (5.1) is equivalent to "$\mathcal{CH}(A)$ assuming $\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z)$". In mathematical logic, a statement of this form is called a **sequent** and is denoted by

$$\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z) \vdash \mathcal{CH}(A) \quad . \tag{5.2}$$

In this sequent, the assumptions $\mathcal{CH}(X), \mathcal{CH}(Y), ..., \mathcal{CH}(Z)$ are called **premises** and the proposition $\mathcal{CH}(A)$ is called the **goal**. So, describing rigorously the possibility of computing values in functions means proving that sequents of the form (5.2) are true. Conversely, a proof of the sequent (5.2) shows the existence of a code expression of type A that may use previously computed values ("premises") of types $X, Y, ..., Z$.