The Science of Functional Programming

A tutorial, with examples in Scala

Sergei Winitzki

The Science of Functional Programming

The Science of Functional Programming

A tutorial, with examples in Scala

by Sergei Winitzki, Ph.D.

Version of March 29, 2024

Published by **lulu.com** in 2024

Copyright © 2018-2024 by Sergei Winitzki

Print on demand at lulu.com

ISBN (e-book): 978-0-359-76877-6
ISBN: 978-0-359-76877-6

Source hash (sha256): 6a64b82cc3328dc54340c5ffa9a25ebe8b323d977ba66c30751147729828be64

Git commit: dbb82af70b73619a6fda542eda59b00d986e0aa1

PDF file built on Fri, 29 Mar 2024 21:18:14 +0100 by pdfTeX 3.141592653-2.6-1.40.25 (TeX Live 2023) on Darwin

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License" (Appendix F on page 1257).

A *Transparent* copy of the source code for the book is available at `https://github.com/winitzki/sofp` and includes LyX, LaTeX, graphics source files, and build scripts. A full-color hyperlinked PDF file is available at `https://github.com/winitzki/sofp/releases` under "Assets". The source code may be also included as a "file attachment" named `sofp-src.tar.bz2` within a PDF file. To extract, run '`pdftk sofp.pdf unpack_files output .`' and then '`tar jxvf sofp-src.tar.bz2`'. See the file `README_build.md` for build instructions.

This book is a pedagogical in-depth tutorial and reference on the theory of functional programming (FP) as it was practiced at the beginning of the XXI century. Starting from issues found in practical coding, the book builds up the theoretical intuition, knowledge, and techniques that programmers need for rigorous reasoning about types and code. Examples are given in Scala, but most of the material applies equally to other FP languages.

The book's topics include working with FP-style collections; reasoning about recursive functions and types; the Curry-Howard correspondence; laws, structural analysis, and code for functors, monads, and other typeclasses based on exponential-polynomial data types; techniques of symbolic derivation and proof; free typeclass constructions; and parametricity theorems.

Long and difficult, yet boring explanations are logically developed in excruciating detail through 1882 Scala code snippets, 191 statements with step-by-step derivations, 103 diagrams, 218 examples with tested Scala code, and 297 exercises. Discussions build upon each chapter's material further.

Beginners in FP will find tutorials about the map/reduce programming style, type parameters, disjunctive types, and higher-order functions. For more advanced readers, the book shows the practical uses of the Curry-Howard correspondence and the parametricity theorems without unnecessary jargon; proves that all the standard monads (e.g., `List` or `State`) satisfy the monad laws; derives lawful instances of `Functor` and other typeclasses from types; shows that monad transformers need 18 laws; and explains the use of parametricity for reasoning about the Church encoding and the free typeclasses.

Readers should have a working knowledge of programming; e.g., be able to write code that prints the number of distinct words in a sentence. The difficulty of this book's mathematical derivations is at the level of undergraduate multivariate calculus, similar to that of multiplying matrices or simplifying the expressions:

$$\frac{1}{x-2}-\frac{1}{x+2}\quad\text{and}\quad\frac{d}{dx}\left((x+1)f(x)e^{-x}\right)\quad.$$

The author received a Ph.D. in theoretical physics. After a career in academic research, he works as a software engineer.

Contents

Preface

This book is a reference text and a tutorial that teaches functional programmers how to reason mathematically about types and code, in a manner directly relevant to software practice. The material ranges from introductory (Part I) to advanced (Part III). The book assumes a certain amount of mathematical experience, at about the level of undergraduate algebra or calculus, as well as some experience writing code in general-purpose programming languages.

The vision of this book is to explain the mathematical theory that guides the practice of functional programming. So, all mathematical developments in this book are motivated by practical programming issues and are accompanied by Scala code illustrating their usage. For instance, the laws for standard typeclasses (functors, monads, etc.) are first motivated heuristically through code examples. Then the laws are formulated as mathematical equations and proved rigorously.

To achieve a clearer presentation of the material, the book uses certain non-standard notations (see Appendix A on page 1149) and terminology (Appendix B on page 1159). The presentation is self-contained, defining and explaining all required techniques, notations, and Scala features. Although the code examples are in Scala, the material in this book also applies to many other functional programming languages.

All concepts and techniques are illustrated by examples and explained as simply as possible ("but not simpler", as Einstein said). Exercises should be attempted after absorbing the preceding material.

A software engineer needs to learn only those few fragments of mathematical theory that answer questions arising in the programming practice. So, this book keeps theoretical material at the minimum: *vita brevis, ars longa*. The scope of the required mathematical knowledge is limited to first notions of set theory, formal logic, and category theory. Concepts such as functors or natural transformations arise organically from the practice of reasoning about code and are first introduced without reference to category theory.

This book is *not* an introduction to current theoretical research in functional programming. Instead, the focus is on material known to be practically useful. The book organically develops the scope of theoretical concepts that help programmers write code or answer practical questions about code. That includes constructions such as "filterable functors" and "applicative contrafunctors" but excludes a number of theoretical developments that do not appear to have significant applications. For instance, this book does not talk about introduction/elimination rules, strong normalization, complete partial orders, domain theory, model theory, adjoint functors, co-ends, pullbacks, or topoi.

The first part of the book introduces functional programming. Readers already familiar with functional programming could skim the glossary (Appendix B on page 1159) for unfamiliar terminology and then start reading Chapter 5.

Chapters 5–6 begin using the code notation, such as Eq. (6.15). If that notation still appears hard to follow after going through Chapters 5–6, readers will benefit from working through Chapter 7, which summarizes the code notation more systematically and clarifies it with additional examples.

All code examples have been tested to work but are intended only for explanation and illustration. As a rule, the code is not optimized for performance.

The author thanks Joseph Kim and Jim Kleck for doing some of the exercises and reporting some errors in earlier versions of this book. The author also thanks Bill Venners for many helpful comments on the draft, and Harald Gliebe, Andreas Röhler, and Philip Schwarz for contributing corrections to the text via `github`. The author is grateful to Frederick Pitts and several anonymous `github` contributors who reported errors in the draft and made helpful suggestions, and to Barisere Jonathan for valuable assistance with setting up automatic builds.

Formatting conventions used in this book

- Text in boldface indicates a new concept or term that is being defined at that place in the text. Italics means logical emphasis. Example:

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

- Equations are numbered per chapter: Eq. (1.3). Statements, examples, and exercises are numbered per subsection: Example 1.4.1.1 is in subsection 1.4.1, which belongs to Chapter 1.

- Scala code is written inline using a small monospaced font: `val a = "xyz"`. Longer code examples are written in separate code blocks and may also show the Scala interpreter's output for certain lines:

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

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

- In the introductory chapters, type expressions and code examples are written in the syntax of Scala. Starting from Chapters 4–5, the book introduces a mathematical notation for types: for example, the Scala type expression `((A, B)) => Option[A]` is written as $A \times B \to \mathbb{1} + A$. Chapters 4–7 also develop a more concise notation for code. For example, the functor composition law (in Scala: `_.map(f).map(g) == _.map(f andThen g)`) is written in the code notation as:

$$f^{\uparrow L} \fatsemi g^{\uparrow L} = (f \fatsemi g)^{\uparrow L} \quad ,$$

where L is a functor and $f^{:A\to B}$ and $g^{:B\to C}$ are arbitrary functions of the specified types. The notation $f^{\uparrow L}$ denotes the function f lifted to the functor L and replaces Scala's syntax `x.map(f)` where `x` is of type `L[A]`. The symbol $\fatsemi$ denotes the forward composition of functions (Scala's method `andThen`). Appendix A on page 1149 summarizes this book's notation for types and code.

- Frequently used methods of standard typeclasses, such as Scala's `flatten`, `flatMap`, etc., are denoted by shorter words and are labeled by the type constructor they belong to. For instance, the methods `pure`, `flatten`, and `flatMap` for a monad M are denoted by pu_M, ftn_M, and flm_M when writing code formulas and proofs of laws.

- Derivations are written in a two-column format. The right column contains formulas in the code notation. The left column gives an explanation or indicates the property or law used to derive the expression at right from the previous expression. A green underline shows the parts of an expression that will be rewritten in the *next* step:

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

When the two-column presentation becomes too wide to fit the page, the explanations are placed before the next step's line:

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

A green underline is sometimes also used at the last step of a derivation, to indicate the sub-expression that resulted from the most recent rewriting. Other than providing hints to help clarify the steps, the green text and the green underlines *play no role* in symbolic derivations.

- The symbol $\square$ is used occasionally to indicate more clearly the end of a definition, a derivation, or a proof.

Part I

Introductory level

1 Mathematical formulas as code. I. Nameless functions

1.1 Translating mathematics into code

1.1.1 First examples

We begin by implementing some computational tasks in Scala.

Example 1.1.1.1 Find the product of integers from 1 to 10 (the **factorial** of 10, usually denoted by 10!).

Solution First, we write a mathematical formula for the result:

$$10! = 1 * 2 * ... * 10 \quad , \qquad \text{or in mathematical notation :} \quad 10! = \prod_{k=1}^{10} k \quad .$$

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

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

The syntax `(1 to 10)` produces a sequence of integers from `1` to `10`. The `product` method computes the product of the numbers in the sequence.

The code `(1 to 10).product` is an **expression**, which means that (1) the code can be evaluated and yields a value, and (2) the code can be used inside a larger expression. For example, we could write:

```
scala> 100 + (1 to 10).product + 100   // The code `(1 to 10).product` is a
    sub-expression.
res0: Int = 3629000

scala> 3628800 == (1 to 10).product
res1: Boolean = true
```

The Scala interpreter indicates that the result of `(1 to 10).product` is a value 3628800 of type `Int`. If we need to define a name for that value, we use the "`val`" syntax:

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

Example 1.1.1.2 Define a function to compute the factorial of an integer argument n.

Solution 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; here, we wrote `n: Int`. In the usual mathematical notation, types of function arguments are either not written at all, or written separately from the formula:

$$f(n) = \prod_{k=1}^{n} k \quad , \qquad \forall n \in \mathbb{N} \quad . \tag{1.1}$$

Equation (1.1) indicates that n must be from the set of positive integers, denoted by $\mathbb{N}$ in mathematics. This is similar to specifying the type `(n: Int)` in the Scala code. So, the argument's type in the code specifies the **domain** of a function (the set of admissible values of a a function's argument).

Having defined the function `f`, we can now apply it to an integer value `10` (or, as programmers say, "call" the function `f` with argument `10`):

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

It is a **type error** to apply `f` to a non-integer value:

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

1.1.2 Nameless functions

Both the code written above and Eq. (1.1) involve *naming* the function as "f". Sometimes a function does not really need a name, — say, if the function is used only once. "Nameless" mathematical functions may be denoted using the symbol $\to$ (pronounced "maps to") like this:[1]

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

So, a mathematical notation for the nameless factorial function is:

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

[1]In mathematics, an often used symbol for "maps to" is $\mapsto$, but this book uses a simpler arrow symbol ($\to$) that is visually similar.

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** variable,[2] while `(1 to n).product` is the function's **body**. The function arrow (`=>`) separates the argument variable from the body.[3]

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 `fac` takes an integer (`Int`) argument and returns an integer result value. What is the value of the function `fac` *itself*? As we have just seen, the standard Scala interpreter prints `<function1>` as the "value" of `fac`. Another Scala interpreter called `ammonite`[4] prints this:

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

The long number could indicate an address in memory. We may imagine that a "function value" represents a block of compiled code. That code will run and evaluate the function's body whenever the function is applied to an argument.

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

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

Functions can be also used without naming them. We may 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
```

We would rarely write code like this. Instead of creating a nameless function and then applying it right away to an argument, it is easier to evaluate the expression symbolically by substituting `10` instead of `n` in the function body:

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

If a nameless function uses the argument several times, as in this code:

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

[2] In computer science, argument variables are called "parameters" of a function, while an "argument" is a value to which the function is actually applied. This book uses the word "argument" for both, following the mathematical usage.

[3] Some programming languages use the symbols `->` or `=>` for the function arrow; see Table 1.2.

[4] See https://ammonite.io/

we could substitute the argument and eliminate the nameless function:

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

Of course, it is better to avoid repeating the value `12345`. To achieve that, 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 **local-scope** definition.

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

Example 1.1.2.1 Define a function that takes an integer argument n and determines whether n is a prime number.

Solution By definition, n is prime if, for all k between 2 and $n-1$, the remainder after dividing r by k (denoted by $r\%k$) is nonzero. We can write this as a mathematical formula using the "forall" symbol ($\forall$):

$$\text{isPrime}\,(n) = \forall k \in [2, n-1]\,.\,(n\%k) \neq 0 \quad . \tag{1.2}$$

This formula has two parts: first, a range of integers from 2 to $n-1$, and second, a requirement that all these integers k should satisfy the given condition: $(n\%k) \neq 0$. Formula (1.2) is translated into Scala code as:

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

This code looks closely similar to the mathematical notation, except for the arrow after k that introduces a nameless function (`k => n % k != 0`). We do not need to specify the type `Int` for the argument `k` of that nameless function. 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` because types must match.

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

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

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

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

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

In Scala, it is strongly recommended (although often not mandatory) to specify the return types of named functions. The required syntax looks like this:

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

1.1.3 Nameless functions and bound variables

The code for `isPrime` differs from the mathematical formula (1.2) in two ways.

One difference is that the interval $[2, n-1]$ is in front of `forall`. Another is that the Scala code uses a nameless function `(k => n % k != 0)`, while Eq. (1.2) does not seem to use such a function.

To understand the first difference, we need to keep in mind that 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 **method** syntax `x.f(z)`, and the equivalent **infix** syntax `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 widely used when it is easier to read: for instance, we write $x + y$ rather than something like $plus\,(x, y)$.

A single-argument function could be also defined as a method, and then the syntax is `x.f`, as in the expression `(1 to n).product` shown before.

The 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 method syntax, we could define a function `forAll` with two arguments and write code like this:

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

This would bring the syntax closer to Eq. (1.2). 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 formula:

$$\forall k \in [2, n-1]\,.\,(n\%k) \neq 0 \tag{1.3}$$

does not seem to define such a function but defines the symbol k that goes over the range $[2, n-1]$. The variable k is then used for writing the predicate $(n\%k) \neq 0$.

Let us investigate the role of k more closely. The mathematical variable k is accessible *only inside* the expression "$\forall k. ...$" and makes no sense outside that expression. This becomes clear by looking at Eq. (1.2): The variable k is not present in the left-hand side and could not possibly be used there. The name "k" is accessible 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 "$n\%k$".

So, the mathematical notation in Eq. (1.3) says two things: First, we use the name k for integers from 2 to $n-1$. Second, for each of those k we evaluate the expression $(n\%k) \neq 0$, which can be viewed as a certain *function of* k that returns a `Boolean` value. Translating the mathematical notation into code, it is natural to use the nameless function $k \to (n\%k) \neq 0$ and to write Scala code applying this nameless function to each element of the range $[2, n-1]$ and checking 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. (1.2), the argument `k` of the nameless Scala function `k => n % k != 0` is defined 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 $\forall k.\,(n\%k) \neq 0$, the variable k is bound (defined and only visible within that expression's scope) but the variable n is free: it must be defined somewhere outside the expression $\forall k.\,(n\%k) \neq 0$.

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. (1.2) an equivalent definition:

$$\text{isPrime}\,(n) = \forall z \in [2, n-1]\,.\,(n\%z) \neq 0 \quad .$$

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

The argument `z` in the nameless function `z => n % z != 0` is a bound variable: it may be renamed without changing any code outside that function. But `n` is a free variable within `z => n % z != 0` (it is not defined inside that function, so it must be defined outside). If we wanted to rename `n` in the sub-expression `z => n % z != 0`, we would also need to change all the code that involves the variable `n` outside that sub-expression, or else the program would become incorrect.

Mathematical formulas use bound variables in 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 bound variables present in the mathematical notation. For each bound variable, we create a nameless function whose argument is that variable, e.g., `k => p(k)` or `k => f(k)` for the examples just shown. Then our code will correctly reproduce the behavior of bound variables in mathematical expressions.

As an example, the mathematical formula $\forall k \in [1, n]\,.\,p(k)$ 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 a simplification trick to this code. The nameless function $k \to p(k)$ 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 $x \to f(x)$ to just f is always possible for functions f of a single argument.[5]

1.2 Aggregating data from sequences

Consider the task of counting the *even* numbers contained 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 using the "sum" operation (denoted by $\sum$):

$$\mathrm{countEven}\,(L) = \sum_{k \in L} \mathrm{isEven}\,(k) \quad ,$$

$$\mathrm{isEven}\,(k) = \begin{cases} 1 & \text{if } (k\%2) = 0 \quad , \\ 0 & \text{otherwise} \quad . \end{cases}$$

Here we defined a helper function `isEven` in order to write more easily a formula for `countEven`. 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 `isEven`; the Scala code can be written in a style quite similar to the mathematical formula:

```
def isEven(k: Int): Int = (k % 2) match {
  case 0 => 1 // First, check if it is zero.
  case _ => 0 // The underscore means "otherwise".
}
```

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

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

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

To compute $\sum_{k \in L} \mathrm{isEven}\,(k)$, we must apply the function `isEven` 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 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:

[5] Certain features of Scala allow programmers to write code that looks like `f(x)` but actually uses an automatic conversion for the argument `x`, default argument values, or implicit arguments. In those cases, replacing the code `x => f(x)` by `f` will fail to compile.

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

The method `map` needs more explanation. This method takes a *function* as its second argument and applies that function to each element of the list. All the results are stored in 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 $x \to x^2 + 100x$. This function will be used repeatedly by `map` to transform each integer from the sequence `List(1, 2, 3)`, creating a new list as a result.

It is equally possible to define the transforming function separately, give it a name, and then use 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)
```

Short functions are often defined inline, while longer functions are defined separately with a name.

A method, such as `map`, can be also used with a "dotless" (**infix**) syntax:

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

If the transforming function is used only once (such as `func1` in the example above), and especially for simple computations such as $x \to x^2 + 100x$, it is easier to use a nameless function.

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

```
def countEven(s: List[Int]) = s.map(isEven).sum
```

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

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

In Scala, methods are often used one after another, as if in a chain. For instance, `s.map(...).sum` means: first apply `s.map(...)`, which returns a *new* list; then apply `sum` to that new list. To make the code more readable, we may 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 multi-line code, we will enclose the code in braces:

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

```
     |   .sum
     | }
def countEven: (s: List[Int])Int

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

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

Note that the Scala interpreter prints the types differently for named functions (i.e., functions declared using `def`). It prints `(s: List[Int])Int` for a function of type `List[Int] => Int`.

1.3 Filtering and truncating a sequence

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` if and only if the predicate returns `true` for all values in the list. The `exists` method returns `true` if and only if the predicate holds (returns `true`) for at least one value in the list. These methods can be written as mathematical formulas like this:

$$\begin{aligned} \text{forall}\,(S,p) &= \forall k \in S.\, \big(p(k) = \text{true}\big) \quad, \\ \text{exists}\,(S,p) &= \exists k \in S.\, \big(p(k) = \text{true}\big) \quad. \end{aligned}$$

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

```
scala> List(1, 2, 3, 4, 5).filter(k => k != 3)          // Exclude the value 3.
res5: List[Int] = List(1, 2, 4, 5)
```

The `takeWhile` method truncates a given list. More precisely, `takeWhile` returns a new list that contains the initial portion of values from the original list for which predicate remains `true`:

```
scala> List(1, 2, 3, 4, 5).takeWhile(k => k != 3)    // Truncate at the value 3.
res6: List[Int] = List(1, 2)
```

In all these cases, the predicate's argument, `k`, will 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 `sum` and `product` are defined for lists of numeric types, such as `Int` or `Float`. The methods `max` and `min` are defined on lists of "orderable" types (including `String`, `Boolean`, and the numeric types). The other methods are defined for lists of all types.

Using these methods, we can solve many problems that involve transforming and aggregating data stored in lists, arrays, sets, and other data structures that work as "containers storing values". In this context, a **transformation** is a function taking a container with values and returning a new container with changed values. (We speak of "transformation" even though the original container *remains unchanged*.) Examples of transformations are `filter` and `map`. An **aggregation** is a function taking a container of values and returning a *single* value. Examples of aggregations are `max` and `sum`.

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

1.4 Examples

1.4.1 Aggregations

Example 1.4.1.1 Improve the code for `isPrime` by limiting the search to $k \leq \sqrt{n}$:

$$\text{isPrime}\,(n) = \forall k \in [2, n-1] \text{ such that if } k * k \leq n \text{ then } (n \% k) \neq 0 \quad .$$

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

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

Example 1.4.1.2 Compute this product of absolute values: $\prod_{k=1}^{10} |\sin(k+2)|$.

Solution

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

Example 1.4.1.3 Compute $\sum_{k\in[1,10];\ \cos k>0} \sqrt{\cos k}$ (the sum goes only over k such that $\cos k > 0$).

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 $\cos k > 0$. Let us check that this is so:

```
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:

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

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

Example 1.4.1.5 Given n, compute the Wallis product[6] truncated up to $\frac{2n}{2n+1}$:

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

Solution 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).toDouble / (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 cosine of `wallis(n)` tends to zero for large n because the limit of the Wallis product is $\frac{\pi}{2}$.

Example 1.4.1.6 Check numerically the following infinite product formula:

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

[6] https://en.wikipedia.org/wiki/Wallis_product

Solution Compute this product up to $k = n$ for $x = 0.1$ with 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 1.4.1.7 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 a test for this function:

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

1.4.2 Transformations

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`, where `t` is of type `List[Int]`:

```
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 Scala compiler deduces from the code that the type of `t` is `List[Int]` because we apply `filter` to 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 $j * j > 2 * k$.

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 also uses a `filter`. The inner expression:

Mathematical notation	Scala code
$x \to \sqrt{x^2+1}$	`x => math.sqrt(x * x + 1)`
$[1, 2, ..., n]$	`(1 to n)`
$[f(1), ..., f(n)]$	`(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`
$\forall k \in [1, ..., n].\, p(k)$ holds	`(1 to n).forall(k => p(k))`
$\exists k \in [1, ..., n].\, p(k)$ 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.

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

computes a list of all j's that satisfy the condition $j * j > 2 * k$. The size of that list is then compared with 3. In this way, we impose 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 task.

1.5 Summary

Functional programs are mathematical formulas translated into code. Table 1.1 summarizes the tools explained in this chapter and gives implementations of some mathematical constructions in Scala. We have also shown methods such as `takeWhile` that do not correspond to widely used mathematical symbols.

What problems can one solve with these techniques?

- Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges, such as $\sum_{k=1}^{n} f(k)$.
- 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 $f(f(f(...f(0)...))) \geq 1000$, where the given function f is applied n times.

- Example 2: Compute a list of partial sums from a given list of integers. For example, the list $[1,2,3,4]$ should be transformed into $[1,3,6,10]$.

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

These computations require a general case of *mathematical induction*. Chapter 2 will explain how to implement these tasks using recursion as well as using library methods such as `foldLeft`.

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. In those cases, the number of iterations is known (or at least bounded) in advance. 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 translate that into Scala code.

An implementation of Example 2 is shown in Section 2.4. This cannot be implemented with operations such as `map` and `filter` because they cannot produce sequences whose next elements depend on previous values.

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

1.6 Exercises

1.6.1 Aggregations

Exercise 1.6.1.1 Define a function that computes a "staggered factorial" (denoted by $n!!$) for positive integers. It is defined as either $1 \cdot 3 \cdot ... \cdot n$ or as $2 \cdot 4 \cdot ... \cdot n$, depending on whether n is even or odd. For example, $8!! = 384$ and $9!! = 945$.

Exercise 1.6.1.2 Machin's formula[7] converges to π faster than Example 1.4.1.5:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad ,$$

$$\arctan \frac{1}{n} = \frac{1}{n} - \frac{1}{3}\frac{1}{n^3} + \frac{1}{5}\frac{1}{n^5} - ... = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} n^{-2k-1} \quad .$$

[7] http://turner.faculty.swau.edu/mathematics/materialslibrary/pi/machin.html

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 12 terms of the series are sufficient for a full-precision `Double` approximation of π.

Exercise 1.6.1.3 Check numerically that $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. First, define a function of n that computes a partial sum of that series until $k = n$. Then compute the partial sum for a large value of n and compare with the limit value.

Exercise 1.6.1.4 Using the function `isPrime`, check numerically the Euler product formula[8] for the Riemann's zeta function $\zeta(4)$. It is known[9] that $\zeta(4) = \frac{\pi^4}{90}$:

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

1.6.2 Transformations

Exercise 1.6.2.1 Define a function `add20` of type `List[List[Int]] => List[List[Int]]` that adds 20 to every element of every inner list. A sample test:

```
scala> add20( 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 i, j, k such that $1 < i < j < k < n$. 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 $1 < i < j < k < n$ such that $f(i)$, $f(j)$, and $f(k)$ are all `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 `at100` of type `List[List[Int]] => List[List[Int]]` that selects only those inner lists whose largest value is at least 100. Test with:

```
scala> at100( 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 performs a "normalization" of a list: it finds the element having the largest absolute value and, if that value is zero, returns the original list; if that value is nonzero, divides all elements by that value and returns a new list. Test with:

```
scala> normalize(List(1.0, -4.0, 2.0))
res0: List[Double] = List(0.25, -1.0, 0.5)
```

[8] http://tinyurl.com/4rjj2rvc
[9] https://tinyurl.com/yxey4tsd

1.7 Discussion

1.7.1 Functional programming as a paradigm

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

The main idea of FP is to write code *as a mathematical expression or formula*. This 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." Like 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 useful for programming tasks because it is backed by the vast experience of working with data over millennia of human history. It took centuries to invent flexible and powerful notation, such as $\sum_{k\in S} p(k)$, and to develop the corresponding rules of calculation. Converting formulas into code, FP capitalizes on the power of these reasoning tools.

As we have seen, the Scala code for certain computational tasks corresponds quite closely to mathematical formulas (although programmers do have to write out some details that are omitted in the mathematical notation). Just as in mathematics, large code expressions may be split into smaller expressions when needed. Expressions can be reused, composed in various ways, and written independently from each other. Over the years, the FP community has developed a toolkit of functions (such as `map`, `filter`, `flatMap`, etc.), which are not standard in mathematical literature but proved to be useful in practical programming.

Mastering FP involves practicing to write programs as "formulas translated into code", building up the specific kind of applied mathematical intuition, and getting familiar with certain concepts adapted to a programmer's needs. The FP community has discovered a number of specific programming idioms founded on mathematical principles but driven by practical necessities of writing software. This book explains the theory behind those idioms, starting from code examples and heuristic ideas, and gradually building up the techniques of rigorous reasoning.

This chapter explored the first significant idiom of FP: iterative calculations performed without loops in the style of mathematical expressions. This technique can be used in any programming language that supports nameless functions.

1.7.2 Iteration without loops

In mathematical notation, iterative computations are written without loops. As an example, consider the formula for the standard deviation (σ) estimated from a data sample $[x_1, ..., x_n]$:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} x_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} x_i\right)^2} \quad .$$

Here the index i goes over the integer range $[1, ..., n]$. And yet no mathematics textbook would define this formula via loops or by saying "now repeat this equation ten times". Indeed, it is unnecessary to evaluate a formula such as x_i^2 ten times, as the value of x_i^2 remains the same every time. It is just as unnecessary to "repeat" a mathematical equation.

Instead of loops, mathematicians write *expressions* such as $\sum_{i=1}^{n} s_i$, where symbols such as $\sum_{i=1}^{n}$ denote certain iterative computations. Such computations are can be defined rigorously using mathematical induction. The FP paradigm has developed rich tools for translating mathematical induction into code. This chapter focuses on methods such as `map`, `filter`, and `sum`. The next chapter shows more general methods for implementing inductive computations. These methods can be combined in flexible ways, enabling programmers to write iterative code without loops. For example, the value σ defined by the formula shown above is computed by code that looks like this:

```
def sigma(xs: Seq[Double]): Double = {
  val n = xs.length.toDouble
  val xsum = xs.sum
  val x2sum = xs.map(x => x * x).sum
  math.sqrt(x2sum / (n - 1) - xsum * xsum / n / (n - 1))
}

scala> sigma(Seq(10, 20, 30))
res0: Double = 10.0
```

The programmer can avoid writing loops because all iterative computations are delegated to functions such as `map`, `filter`, `sum`, and others. It is the job of the library and the compiler to translate those high-level functions into low-level machine code. The machine code *will* likely contain loops, but the programmer does not need to see that machine code or to reason about it.

1.7.3 The mathematical meaning of "variables"

The usage of variables in functional programming is similar 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, 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 $k \to k^2 + k$. 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". 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 consider them in this book.)

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

```
def f(x: Int) = x * x + x // Cannot modify ‘x‘ here.
```

The *type* of each mathematical variable (such as integer, vector, etc.) is also fixed. Each variable is a value from a specific set (e.g., the set of all integers, the set of all vectors, etc.). Mathematical formulas such as $x^2 + x$ do not express any "checking" that x is indeed an integer and not, say, a vector, in the middle of evaluating $x^2 + x$. The types of all variables are checked in advance.

Functional programming adopts the same view: Each argument of each function must have a **type** that represents the set of possible allowed values for that function argument. The programming language's compiler will automatically check the types of all arguments in advance, *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
```

We could also omit the type annotation "`:Int`" and write more concisely:

```
val x = 123
```

Here, it is clear that this `x` is an integer. Nevertheless, it is often helpful to write out the types. If we do so, the compiler will check that the types match correctly and give an error message whenever wrong types are used. For example, a type error is detected when using a `String` instead of an `Int`:

```
scala> val x: Int = "123"
<console>:11: error: type mismatch;
 found   : String("123")
 required: Int
       val x: Int = "123"
                    ^
```

1.7.4 Nameless functions in mathematical notation

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

Nameless functions contain bound variables that are invisible outside the function's scope. This property is directly reflected by the prevailing mathematical conventions. Compare the formulas:

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

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

In programming, one situation when a variable "may be renamed at will" is when the variable represents an argument of a function. We can see 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:

```
integration(0, x, { z => 1.0 / (1 + z) } )
```

Now compare the mathematical notations for integration and for summation: $\int_0^x\frac{dz}{1+z}$ and $\sum_{k=0}^{100}\frac{1}{1+k}$. The integral defines a bound variable z via the special symbol "d", while the summation places a bound variable k in a subscript under $\sum$. The notation could be made more consistent by using nameless functions explicitly, for example like this:

$$\text{denote summation by } \sum_0^x \left(k \to \frac{1}{1+k}\right) \text{ instead of } \sum_{k=0}^x \frac{1}{1+k} \quad ,$$
$$\text{denote integration by } \int_0^x \left(z \to \frac{1}{1+z}\right) \text{ instead of } \int_0^x \frac{dz}{1+z} \quad .$$

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

```
summation(0, x, { y => 1.0 / (1 + y) } )
```

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

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

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

Integration requires longer code since the computations are more complicated. Simpson's rule[10] gives the following formulas for numerical integration:

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

Here is a straightforward line-by-line translation of these formulas into Scala:

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

[10] https://en.wikipedia.org/wiki/Simpson%27s_rule

```
scala> simpson(0, 5, x => x*x*x*x, eps = 0.01)       // The answer is 625.
res0: Double = 625.0000000004167

scala> simpson(0, 7, x => x*x*x*x*x*x, eps = 0.01)   // The answer is 117649.
res1: Double = 117649.00000014296
```

The entire code is one large *expression*, with a few sub-expressions (`s1`, `s2`, etc.) defined for within the **local scope** of the function (that is, within the function's body). The code contains no loops. This is similar to the way a mathematical text would define Simpson's rule. In other words, this code is written in the FP paradigm. Similar code can be written in any programming language that supports nameless functions as arguments of other functions.

1.7.5 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 today's programming languages, we may directly write expressions such as `(x + 123) * y / (4 + x)`. Note that the entire expression does not need to have a name. Parts of that expression (e.g., the sub-expressions `x + 123` or `4 + x`) also do not have separate names. It would be inconvenient if we *needed* to assign a name to each sub-expression. The code for `(x + 123) * y / (4 + x)` would look like this:

```
{
  val r0 = 123
  val r1 = x + r0
  val r2 = r1 * y
  val r3 = 4
  val r4 = r3 + x
  val r5 = r2 / r4     // Do we still remember what `r2` means?
  r5
}
```

This style of programming resembles assembly languages: every sub-expression — that is, every step of every calculation — needs to be written into a separate memory address or a CPU register.

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 also useful to be able to create nameless data structures. For instance, a **dictionary** (also called a "map" or a "hashmap") is created in Scala with this code:

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

This is a nameless expression whose value is a dictionary. In programming languages that do not have such a construction, programmers have to write special code that creates an initially empty dictionary and then fills in one value at a time:

```
// Scala code creating a dictionary:
Map("a" -> 1, "b" -> 2, "c" -> 3)

// Shortest Java code for the same:
new HashMap<String, Integer>() {{
   put("a", 1);
   put("b", 2);
   put("c", 3);
}}
```

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

1.7.6 Historical perspective on nameless functions

What this book calls (for clarity) a "nameless function" is also known as an anonymous function, a function expression, a function literal, a closure, a lambda function, a lambda expression, or just a "lambda".

Nameless functions were first used in 1936 in a theoretical programming language called "λ-calculus". In that language,[11] all functions are nameless and have a single argument. The Greek letter λ is a syntax separator that denotes function arguments in nameless functions. For example, the nameless function $x \to x + 1$ would be written as $\lambda x.\ add\ x\ 1$ in λ-calculus if it had a function *add* for adding integers (but it does not).

In most programming languages that were in use until around 1990, all functions required names. But by 2015, the use of nameless functions in the `map`/`reduce` programming style turned out to be so productive that most newly created languages included nameless functions, while older languages added that feature. Table 1.2 shows the year when various languages supported nameless functions.

[11] Although called a "calculus," it is a (drastically simplified) *programming language*, not related to differential or integral calculus. Practitioners of functional programming do not need to study the theory of λ-calculus. The practically relevant knowledge that comes from λ-calculus will be explained in Chapter 4.

Language	**Year**	**Code for $k \to k+1$**
λ-calculus	1936	$\lambda k.\ add\ k\ 1$
typed λ-calculus	1940	$\lambda k : int.\ add\ k\ 1$
LISP	1958	`(lambda (k) (+ k 1))`
ALGOL 68	1968	`(INT k) INT: k + 1`
Standard ML	1973	`fn (k: int) => k + 1`
Caml	1985	`fun (k: int) -> k + 1`
Erlang	1986	`fun(K) -> K + 1 end`
Haskell	1990	`\ k -> k + 1`
Oz	1991	`fun {$ K} K + 1`
R	1993	`function(k) k + 1`
Python 1.0	1994	`lambda k: k + 1`
JavaScript	1995	`function(k) { return k + 1; }`
Mercury	1995	`func(K) = K + 1`
Ruby	1995	`lambda { |k| k + 1 }`
Lua 3.1	1998	`function(k) return k + 1 end`
Scala	2003	`(k: Int) => k + 1`
F#	2005	`fun (k: int) -> k + 1`
C# 3.0	2007	`delegate(int k) { return k + 1; }`
Clojure	2009	`(fn [k] (+ k 1))`
C++ 11	2011	`[] (int k) { return k + 1; }`
Go	2012	`func(k int) { return k + 1 }`
Julia	2012	`function(k :: Int) k + 1 end`
Kotlin	2012	`{ k: Int -> k + 1 }`
Swift	2014	`{ (k: int) -> int in return k + 1 }`
Java 8	2014	`(int k) -> k + 1`
Rust	2015	`|k: i32| k + 1`

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.

First, we need to become fluent in using tuple types with Scala collections.

2.1 Tuple types

2.1.1 Examples: Using tuples

Many standard library methods in Scala work with tuple types. A simple example of a tuple is a *pair* of values, e.g., 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 the type of this pair.

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 several values called **parts** or **fields** of a tuple. A tuple's parts can have different types, but the type of each part (and the number of parts) is fixed once and for all. 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** such that any part of a tuple can be itself a tuple:

```
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* `=> ...`

Here is an example of a **destructuring definition**:

```
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 `x`, `y`, `z` are also assigned automatically.

In the example above, the left-hand side of the destructuring definition contains a 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 given pattern. (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 for working with tuples. Look at this example:

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

The expression `{ case (a, b, c) => ... }` is called a **case expression**. It performs pattern matching on its argument. The pattern matching will "destructure" (i.e., decompose) a 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, and so the expression evaluates to 6.

Pattern matching is especially convenient for nested tuples. Here is an example where a nested tuple `p` is destructured by pattern matching:

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

```
scala> t1((10, ("result is ", 2)))
res0: String = result is 12
```

The type structure of the argument `(Int, (String, Int))` is visually repeated in the pattern `(x, (str, y))`, making it clear that `x` and `y` become integers and `str` becomes a string after pattern matching.

If we rewrite the code of `t1` 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 that `p._2._1` is a string. It is also harder to modify 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. We only need to change the type and the pattern:

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

It is easy to see that `t4` and `t1` compute the same result. Also, the names of pattern variables may be chosen to get more clarity.

Sometimes we only need to use certain parts of a tuple in a pattern match. The following syntax is used to make that 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 its type.

Scala has a shorter syntax for functions such as `{case (x, y) => y}` that extract elements from tuples. The syntax looks like `(t => t._2)` or equivalently `_._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 = (t => t._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)

scala> q._1(3)
res0: Int = 4
```

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. (As usual, we speak of "data transformation" even though the original list remains unchanged.) 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 }
res1: List[Int] = List(6, 4, 0)

scala> basket.map { case (fruit, count) => count * 2 }.sum
res2: Int = 10
```

In this way, we can use the standard methods such as `map`, `filter`, `max`, `sum` to manipulate sequences of tuples. The names of the pattern variables "`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)
}
res3: List[(String, Boolean)] = List((apples,false), (pears,false),
    (lemons,true))
```

In the Scala syntax, a nameless function written with braces `{ ... }` may 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 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))
```

The same result is obtained by first creating a sequence of key/value *pairs* and then converting that sequence into a dictionary via the method `toMap`:

```
List(("apples", 3), ("oranges", 2), ("pears", 0)).toMap
```

The same method works for other collection types such as `Seq`, `Vector`, and `Array`.

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, the code for creating a dictionary is easier to read:

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

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 subtypes of the common "sequence" type `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 methods that need tuple types, such as `map` and `filter` (when used 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 methods `map` and `toMap` 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 fruitBasket = Map("apples" -> 3, "pears" -> 2, "lemons" -> 0)

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

When using `map` to transform a dictionary into a sequence of pairs, the result is again a dictionary. But when an intermediate result is not a sequence of pairs, we may need to use `toMap`:

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

scala> fruitBasket.map { case (fruit, count) => (fruit, count, count * 2) }.
         map { case (fruit, _, count2) => (fruit, count2 / 2) }.toMap
res2: Map[String,Int] = Map(apples -> 3, pears -> 2, lemons -> 0)
```

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

```
scala> fruitBasket.filter { case (fruit, count) => count > 0 }
res2: Map[String,Int] = Map(apples -> 3, pears -> 2)
```

The methods `zip` and `zipWithIndex` 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 to illustrate a syntax convention of Scala that we will sometimes use.

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

The `zipWithIndex` method creates a sequence of pairs where the second value in the pair is a zero-based index:

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

The method `flatten` converts a nested sequence type, such as `List[List[A]]`, into a simple `List[A]` by concatenating all inner sequences into one:

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

In Scala, sequences and other collections (such as sets and dictionaries) are generally concatenated using the operation `++`. For example:

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

So, one can say that the `flatten` method inserts the operation `++` between all the inner sequences.

By definition, `flatten` removes *only one* level of nesting at the *top* of the data type. If applied to a `List[List[List[Int]]]`, the `flatten` method returns a `List[List[Int]]` with inner lists unchanged:

```
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 method `flatMap` 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 replacing each element by some number (zero or more) of new elements.

At first sight it may be unclear why `flatMap` is useful, as `map` and `flatten` appear to be unrelated. (Should we also combine `filter` and `flatten` into a "`flatFilter`"?) However, we will see later in this book that `flatMap` describes nested iterations and can be generalized to many other data types. This chapter's examples and exercises will illustrate the use of `flatMap` with sequences.

The method `groupBy` 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("xenon", "yogurt", "zebra").groupBy(s => if (s startsWith "y") 1
    else 2)
res12: Map[Int,Seq[String]] = Map(1 -> List(yogurt), 2 -> List(xenon, 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 method `sliding` creates a sequence of sliding windows of a given width:

```
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))
```

After creating a nested sequence, we can apply an aggregation operation to 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, -0.001160413780774369,
    0.003947079736951305, 0.005381273944717851, 0.0018679497047270743)
```

The method `sortBy` sorts a sequence according to a sorting key. The argument of `sortBy` is a *function* that computes the sorting key from a sequence element. This gives us flexibility to elements in a custom way:

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

scala> Seq("xx", "z", "yyy").sortBy(word => word)        // Sort alphabetically.
res1: Seq[String] = List(xx, yyy, z)

scala> Seq("xx", "z", "yyy").sortBy(word => word.length) // Sort by word length.
res2: Seq[String] = List(z, xx, yyy)
```

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("xx", "z", "yyy").sorted
res3: Seq[String] = List(xx, yyy, z)
```

2.1.5 Examples: Tuples and collections

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

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 sequence elements for which the predicate is `true`:

```
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 of `Double` values, compute the sequence of differences $c_i = a_i - b_i$.

Hint: use `zip`, `map`, and assume `as` and `bs` have equal length.

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 one 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))
```

Because `ps.tail` is one element shorter than `ps`, the resulting sequence of pairs is also one element shorter than `ps`. So, 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 $c_i = \max(b_{i-k}, ..., b_{i+k})$, starting at $i = k$.

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

```
val b = List(1, 2, 3, 4, 5)     // An example of a possible sequence ‘b‘.

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

For each 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 `x => x.max` (where `x` will have type `List[Int]`):

```
def c(b: List[Int], k: Int) = b.sliding(2 * k + 1).toList.map(x => x.max)
```

This code can be written more concisely using the syntax:

```
def c(b: List[Int], k: Int) = b.sliding(2 * k + 1).toList.map(_.max)
```

because, in Scala, `_.max` is the same as the nameless function `x => x.max`. Test this:

```
scala> c(b = List(1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1), k = 1)   // Write the
    argument names for clarity.
res0: Seq[Int] = List(3, 4, 5, 6, 6, 6, 5, 4, 3)
```

Example 2.1.5.6 Create a 10×10 multiplication table as a dictionary having the 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 iterate over a sequence of values of `x`; and for each `x`, we then need to 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 => (1, y)` that converts a number into a tuple:

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

The curly braces in `{y => (1, y)}` are only for clarity. We could also use round parentheses and write `List(1, 2, 3).map(y => (1, y))`.

Now, we need to have `(x, y)` instead of `(1, y)` in the argument of `map`, where `x` iterates over `List(1, 2, 3)` in the outside scope. Using this `map` operation, we obtain:

```
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. The dictionary needs to map each *pair* of integers `(x, y)` to `x * y`. To create that dictionary, we will apply `toMap` to a sequence of pairs `(key, value)`, which in our case needs to be of the form of a nested tuple `((x, y), x * y)`. To achieve this, we use `map` with a function that computes the product and creates these nested tuples:

```
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))
```

Applying `toMap`, we convert this list of tuples to a dictionary. Also, for better readability, we use Scala's pair syntax, `key -> value`, which is equivalent to writing the tuple `(key, value)`:

```
(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, x_i^2, $\cos x_i$, $\sin x_i$. Hint: use `flatMap` and `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> val xs = List(0.1, 0.5, 0.9)  // An example list of ‘Double‘ values.
xs: List[Double] = List(0.1, 0.5, 0.9)

scala> xs.map { x => Seq(x, x * x, math.cos(x), math.sin(x)) }
res0: List[Seq[Double]] = List(List(0.1, 0.010000000000000002,
    0.9950041652780258, 0.09983341664682815), List(0.5, 0.25,
    0.8775825618903728, 0.479425538604203), List(0.9, 0.81, 0.6216099682706644,
    0.7833269096274834))
```

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

```
scala> res0.flatten
res1: List[Double] = List(0.1, 0.010000000000000002, 0.9950041652780258,
    0.09983341664682815, 0.5, 0.25, 0.8775825618903728, 0.479425538604203, 0.9,
    0.81, 0.6216099682706644, 0.7833269096274834)
```

It remains to take the maximum of the resulting numbers:

```
scala> res1.max
res2: Double = 0.9950041652780258
```

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

```
xs.flatMap { x => Seq(x, x * 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.

Solution Iterating over a dictionary looks like iterating over a list of `(key, value)` pairs. The result is converted to a `Map` automatically.

```
dict.map { case (name, addr) => (addr, name) } // This has type Map[String,
    String].
```

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) }
```

The body of the function `rev` remains the same as in Example 2.1.5.8; only the type signature changes. This is because the function `rev` works in the same way for dictionaries of any type. For this reason, it was easy for us to change the specific type `String` into type parameters in that 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 some "words", compute a sequence of type `Seq[(Seq[String], Int)]`, where each inner sequence should contain all the words having the same length, paired with the integer value showing that length. The resulting sequence must be ordered by increasing length of words. So, the input `Seq("the", "food", "is", "good")` should produce:

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

Solution 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. To group by word length (computed via the method `length`), we write:

```
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: Map[Int,Seq[String]] = Map(2 -> List(is), 4 -> List(food, good), 3 ->
    List(the))
```

This is 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 can be written somewhat shorter if we use the code `_._1` (equivalent to `x => x._1`) for selecting the first parts from pairs and `swap` for swapping the two elements of a 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 type parameters in collections

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 types. Writing down the intermediate types will help us write correct code.

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 having the type `(Int, Seq[String])`. Then the pattern variables `len` and `words` must have types `Int` and `Seq[String]` respectively.

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]            // After groupBy() by word length, will have type:
Map[Int, Seq[String]]         // To sort by word length, convert to a sequence:
Seq[ (Int, Seq[String]) ]     // Sort by the ‘Int‘ value; type is unchanged:
Seq[ (Int, Seq[String]) ]     // It remains to swap the parts of the tuples:
```

```
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 confidence about code correctness.

2.1.7 Exercises: Tuples and collections

Exercise 2.1.7.1 Find all integer pairs i, j where $0 \leq i \leq 9$ and $0 \leq j \leq 9$ and $i + 4 * j > i * j$.

Hint: use `flatMap` and `filter`.

Exercise 2.1.7.2 Find all integer triples i, j, k where $0 \leq i \leq 9, 0 \leq j \leq 9, 0 \leq k \leq 9$, and $i + 4 * j + 9 * k > i * j * k$.

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` if an `Int` value is followed by a larger value. For example, the input `Seq(1, 3, 2, 4)` must be converted into `Seq((1,true),(3,false),(2,true),(4,false))`. The last value (here, 4) has no following value and is always paired with `false`.

Exercise 2.1.7.5 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 2.1.7.6 Write the solution of Exercise 2.1.7.5 as a function with type parameter `A` instead of the fixed type `String`. The type signature and a sample test:

```
def reorder[A](p: Seq[A], q: Seq[Int]): Seq[A] = ???    // In Scala, ??? means
    "not yet implemented".

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

Exercise 2.1.7.7 Given `p:Seq[String]` and `q:Seq[Int]` of *equal* length and assuming that values in `q` do not repeat, compute a `Map[Int, String]` mapping numbers from `q` to the corresponding strings from `p`.

Exercise 2.1.7.8 Write the solution of Exercise 2.1.7.7 as a function with type parameters `P` and `Q` instead of the fixed types `Int` and `String`. The function's arguments should be of types `Seq[Q]` and `Seq[P]`, and the return type should be `Map[P, Q]`. Run some tests using types `P = Double` and `Q = Set[Boolean]`.

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. So, for the input:

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

the output must be: `Map("apple" -> 10, "pear" -> 3, "lemon" -> 2)`.

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

Exercise 2.1.7.10 Given a `Seq[Seq[Int]]`, compute a new `Seq[Seq[Int]]` where each new inner sequence contains the 3 largest elements from the corresponding old inner sequence, sorted in descending order (or fewer than 3 elements if the old inner sequence is shorter). So, for the input:

```
Seq(Seq(0, 50, 5, 10, 30), Seq(10, 100), Seq(1, 2, 200, 20))
```

the output must be:

```
Seq(Seq(50, 30, 10), Seq(100, 10), Seq(200, 20, 2))
```

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`. This is the set of all pairs `(x, y)` where `x` is an element from `p` and `y` is an element 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]] = ???
// On day 1, Tarski paid 10 and Gödel paid 20. On day 2, Gentzen paid 50, etc.
scala> payments(Seq(Map("Tarski" -> 10, "Gödel" -> 20), Map("Gentzen" -> 50),
    Map("Tarski" -> 50, "Church" -> 100), Map("Banach" -> 15, "Gentzen" -> 35)))
res0: Map[String, Seq[Int]] = Map(Genzten -> List(50, 35), Church -> List(100),
    Banach -> List(15), Tarski -> List(10, 50), Gödel -> List(20))
```

Hint: use `flatMap`, `groupBy`, `map` 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 certain tasks.

A simple computation that is impossible to do using `map` is obtaining the sum of a sequence of numbers. The standard library method `sum` already does this; but we cannot re-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 easily solved 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
```

Note that the same task for integer numbers (instead of floating-point numbers) *can* be implemented via `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, 10, 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
```

For this task, the required 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` or `zip` cannot compute a sequence whose next elements depend on previous elements and the dependence is described by some 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 a sequence to a single value (e.g., an aggregation function `f: Seq[Int] => Int`) via 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()`. The standard method `isEmpty` can be used to detect empty sequences. In case the function `f` 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 :+ x`. So, we need to specify how to compute `f(xs :+ 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.

With this approach, the inductive definition of the method `sum` looks like this: The base case is that the sum of an empty sequence is 0. That is, `Seq().sum == 0`. The inductive step says that when the result `xs.sum` is already known for a sequence `xs`, and we have a sequence that has one more element `x`, then the new result is equal to `xs.sum + x`. In code, this is `(xs :+ x).sum == xs.sum + x`.

The inductive definition of the function `digitsToInt` goes like this: The base case is an empty sequence of digits, `Seq()`, and the result is `0`. This is a convenient base case even if we never need to apply `digitsToInt` to an empty sequence. The inductive step: If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs :+ x` with one more digit `x`, then:

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

Let us write inductive definitions for the methods `length`, `max`, and `count`.

The method `length` Base case: The length of an empty sequence is zero, so we write: `Seq().length == 0`.

Inductive step: if `xs.length` is known then `(x +: xs).length == xs.length + 1`.

The method `max` The maximum element of a sequence is undefined for empty sequences.

Base case: for a one-element sequence, `Seq(x).max == x`.

Inductive step: if `xs.max` is known then `(x +: xs).max == math.max(x, xs.max)`.

The method `count` computes the number of a sequence's elements satisfying a predicate `p`.

Base case: for an empty sequence, `Seq().count(p) == 0`.

Inductive step: if `xs.count(p)` is known then `(x +: xs).count(p) == xs.count(p) + c`, where we define `c = 1` when `p(x) == true` and `c = 0` otherwise.

When a function is defined by induction, proving a property of that function will usually involve a "proof by induction". As an example, let us prove that `(xs ++ ys).length = xs.length + ys.length`. We use induction on the length of the sequence `xs`. In the base case, we need to prove that the property holds for the base case of the function's definition. In the base case, we need to prove that the property holds for an empty sequence `xs` (and an arbitrary sequence `ys`). To verify the base case, we write: `(Seq() ++ ys).length == ys.length`. In the inductive step of the proof, we assume that the property already holds for some `xs` and `ys` and prove that the property will then hold for `x +: xs` instead of `xs`. To verify that, we use the associativity law of the concatenation operation (to be proved in Statement 8.5.2.1), which allows us to write: `(x +: xs) ++ ys == x +: (xs ++ ys)`. Then:

```
((x +: xs) ++ ys).length     // Expect to equal (x +: xs).length + ys.length
  == (x +: (xs ++ ys)).length
  == 1 + (xs.length + ys.length)
  == (x +: xs).length + ys.length
```

In this way, we show that the property holds for `x +: xs` and `ys` assuming it holds for `xs` and `ys`.

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**. Recursion may be used to implement functions defined by induction.

When the body of a recursive function is evaluated, it may repeatedly call itself with different arguments until a result value can be computed *without* any recursive calls. The repeated recursive calls correspond to inductive steps, and the last call corresponds to the base case of the inductive definition. It is an error (an infinite loop) 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 have the base case or the inductive step. As an example, let us define `sum` by recursion. The base case returns `0`, while the inductive step returns a value computed from the recursive call. Look at this code:

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

The `if/else` expression separates the base case from the inductive step. In the inductive step, it is convenient to split the given sequence `s` into its first element `x`, or the "head" of `s`, and the remainder ("tail") sequence `xs`. So, we split `s` as `s = x +: xs` rather than as `s = xs :+ x`.[1]

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 = x +: xs` or as `s = xs :+ x`, depending on the task at hand.

Let us implement `digitsToInt` according to the inductive definition shown in Section 2.2.1:

```
def digitsToInt(s: Seq[Int]): Int = if (s.isEmpty) 0 else {
  val x = s.last                  // To split s = xs :+ 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` into `xs :+ x` and not into `x +: xs`. The reason is that digits increase their numerical value from right to left, so the correct result is computed if we split `s` into `xs :+ x` and multiply `digitsToInt(xs)` by 10 before adding `x`.

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 a 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 all inductively defined functions will use the same code for checking the base case and 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`:

[1] It is easier to remember the meaning of `x +: xs` and `xs :+ x` if we note that the *col*on (`:`) always points to the *col*lection (`xs`) and the plus sign (`+`) to a single element (`x`) that is being added.

```
def lengthS(s: Seq[Int]): Int = if (s.isEmpty) 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)
...
```

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

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

The code of `lengthS` will repeat the inductive step, that is, the "`else`" part of the "`if/else`", about 100000 times. Each time, the intermediate sub-expression with nested computations `1 + (1 + (...))` will get larger. That sub-expression needs to be held somewhere in memory until the function body goes into the base case, with no more recursive calls. When that happens, the intermediate sub-expression will contain about 100000 nested function calls still waiting to be evaluated. A special area of memory called **stack memory** is dedicated to storing the arguments for all not-yet-evaluated nested function calls. 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 crashes the program.

One way to avoid stack overflows 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, rather than placed inside other computations. Here is an example of tail-recursive code:

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

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 a recursive call to `lengthT(...)`. In the code of `lengthT`, recursive calls never occur within any sub-expressions.

It is not a problem that the recursive call to `lengthT` has some sub-expressions such as `res + 1` 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`. 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 will always use tail recursion when possible. Additionally, Scala has a feature for verifying 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.

The code of `lengthT` with a `tailrec` annotation looks like this:

```
import scala.annotation.tailrec

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

Let us trace the evaluation of this function on an example:

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

All sub-expressions such as `1 + 1` and `2 + 1` are computed *before* recursive calls 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` into 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 (`res + 1`) 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`.

It appears useful to define the helper function (`lengthT`) separately. Then `length` will just call `lengthT` and specify the initial value of the accumulator argument. To emphasize that `lengthT` is a helper function that is only used by `length` to achieve tail recursion, we define `lengthT` as a nested function inside the code of `length`:

```
import scala.annotation.tailrec

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

When `length` is implemented like that, users will not be able to call `lengthT` directly, because `lengthT` 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.isEmpty) res
  else length(s.tail, res + 1)
```

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 but different numbers of arguments:

```
def f(x: Int, y: Boolean) = ...     // Define the function body here.
def f(x: Int): Int = f(Int, false)  // Call the function defined above.
```

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

The accumulator trick works in a large number of cases, but it may be not obvious how to introduce the accumulator argument, what its initial value must be, and how to define the inductive step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the special mathematical property of the expression being computed:

$$1 + (1 + (1 + (... + 0))) = (((0 + 1) + 1) + ...) + 1 \quad .$$

This equation follows from the **associativity law** of addition. So, the computation can be rearranged to group all additions to the left. The accumulator value is the result of adding up to some number of parentheses. In code, it means that intermediate expressions are fully computed before making recursive calls. So, recursive calls always occur outside all other sub-expressions — that is, in tail positions. There are no sub-expressions that need to be stored on the stack until all the recursive calls are complete.

However, not all computations can be rearranged in that way. Even if a code rearrangement exists, it may not be immediately obvious how to find it.

An example is a tail-recursive version of the function `digitsToInt` from the previous subsection, where the sub-expression `digitsToInt(xs) * 10 + x` was a non-tail-recursive call. To transform the code into a tail-recursive form, we need to rearrange the computation:

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

so that the multiplications 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.isEmpty) res
  else fromDigits(s.tail, 10 * res + s.head)

scala> fromDigits(Seq(1, 2, 3, 4))
res0: Int = 1234
```

Despite a similarity between this code and the code of `digitsToInt` from the previous subsection, the implementation of `fromDigits` cannot be directly derived from the inductive definition of `digitsToInt`. We need a separate proof that `fromDigits(s, 0)` computes the same result as `digitsToInt(s)`. This can be proved by using the following property:

Statement 2.2.3.1 For any `s: Seq[Int]` and `r: Int`, the following equation holds:

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

Proof We use induction on the length of `s`. To shorten the proof, denote sequences by $[1, 2, 3]$ 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 the length of a sequence s by $|s|$, 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. For 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 write the inductive step (we will use the symbol $\overset{?}{=}$ to denote equations we still need to prove):

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

We will transform the left-hand side and the right-hand side separately, hoping to 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 use a property of decimal notation: a digit x in front of n other digits has the value $x * 10^n$. This property can be formulated as an equation:

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

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

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

We have successfully 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). *That* proof also uses induction. Begin 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 inductive 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:

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

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

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

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

2.2.4 Implementing general aggregation (`foldLeft`)

An **aggregation** converts a sequence of values into a single value. In general, the type of the result may be different from the type of sequence elements. To describe that 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, we have `f(Seq()) = b0`, where `b0: B` is a given value.
- (Inductive step.) Assuming that `f(xs) = b` is already computed, we define `f(xs :+ x) = g(x, b)` where `g` is a given function with type signature `g: (A, B) => B`.

The code implementing `f` is written using recursion:

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

We can now refactor this code into a generic utility function, by turning `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.isEmpty) 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.take(s.length - 1) :+ s.last` 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 tail-recursive:

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

We call this function 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`, `max`, and many other aggregation functions. The method `leftFold` already contains all the code necessary to set up the base case and the inductive 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 recursive 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

We implemented `leftFold` only as an illustration. Scala's library has a method called `foldLeft` implementing the same logic using a slightly different type signature. To see this difference, compare the implementation of `sum` using our `leftFold` function and using the standard `foldLeft` method:

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

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

The syntax of `foldLeft` makes it more convenient to use a nameless function as the updater argument of `foldLeft`, since curly braces separate that argument from others. We will use the standard `foldLeft` method from now on.

In general, the type of the accumulator value can be different from the type of the sequence elements. An example is an implementation of `count`:

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

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

Since `foldLeft` is tail-recursive, stack overflows will not occur even with long sequences. The method `foldLeft` is available in the Scala library for all collections, including dictionaries and sets.

It is important to gain experience using the `foldLeft` method. The Scala library contains several other methods similar to `foldLeft`, such as `foldRight`, `fold`, and `reduce`. In the following sections, we will mostly focus on `foldLeft` because the other fold-like operations are similar.

2.2.5 Examples: Using `foldLeft`

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 Begin by writing an inductive formulation of the `max` function for sequences. 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 :+ x` is the maximum of `b` and `x`. So, the code is:

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

Example 2.2.5.2 For a given non-empty sequence `xs: Seq[Double]`, compute the minimum, the maximum, and the mean as a tuple $(x_{\text{min}}, x_{\text{max}}, x_{\text{mean}})$. The sequence should be traversed only once; i.e., the entire code must be `xs.foldLeft(...)`, using `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, Int)`. To avoid repeating a long type expression, we can define a type alias for it, say, `D4`:

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

The updater updates 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, 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 2.2.5.3 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. 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* that 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:

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

Example 2.2.5.4 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 2.2.5.5 Implement the function `digitsToDouble` using `foldLeft`. The argument is of type `Seq[Char]`. As a test, `digitsToDouble(Seq('3','4','.','2','5'))` must evaluate to `34.25`. Assume that all input characters are either digits or a dot (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 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.

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

While the dot character was not yet seen, 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:

$$g(n,c) = \begin{cases} n * 10 + c & \text{if the digit is before the dot} \\ n + c/f & \text{if after the dot, where } f = 10, 100, 1000, ... \text{ for each new digit} \end{cases}$$

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 (num, flag, factor) =>
    if (c == '.') (num, true, factor) // Set flag to ‘true‘ after seeing a dot.
    else {
      val digit = c.asDigit        // Convert a character to decimal digit.
      if (flag) (num + digit / factor, flag, factor * 10) // After the dot.
      else (num * 10 + digit, flag, factor)              // Before the dot.
    }
  }
```

Now we can implement `digitsToDouble` like this:

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

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

The result of calling `d.foldLeft` is a tuple `(num, flag, factor)`, in which only the first part, `num`, is needed. In Scala's pattern matching syntax, the underscore (_) denotes pattern variables whose values are not needed in the code. We could get the first part using the accessor method `._1`, but the code will be more readable if we show all parts of the tuple `(num, _, _)`.

Example 2.2.5.6 Implement a function `toPairs` that converts a sequence of type `Seq[A]` to a sequence of pairs, `Seq[(A, A)]`, by putting together the adjacent elements pairwise. If the initial sequence has an odd number of elements, a given default value of type `A` is used to fill the last pair. The required type signature and an example test:

```
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 accumulate a sequence of pairs, and each pair needs two values. However, we iterate over values in the input sequence one by one. So, a new pair can be made 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 "holdover": it holds the previous element for odd-numbered iterations and is empty for even-numbered iterations. Initially, the accumulator should be empty. An example evaluation is:

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:

```
type Acc = (Seq[(A, A)], Seq[A])     // Type alias, for brevity.
def updater(acc: Acc, x: A): Acc = acc match {
    case (result, Seq())      => (result, Seq(x))
    case (result, Seq(prev))  => (result :+ ((prev, x)), Seq())
  }
```

We will call `foldLeft` with this updater and then perform some post-processing to make sure we create the last pair in case the last iteration is odd-numbered, i.e., when the "holdover" 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 :+ ((prev, x)), Seq())
  }
  val (result, holdover) = xs.foldLeft(init)(updater)
  holdover match {    // May need to append the last element to the result.
    case Seq()    => result
    case Seq(x)   => result :+ ((x, default))
  }
}
```

This code shows examples of partial functions that are applied safely. One of these partial functions is used in this sub-expression:

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

This code works when `holdover` is empty or has length 1 but fails for longer sequences. In the implementation of `toPairs`, the value of `holdover` will always be a sequence of length at most 1, so it is safe to use this partial function.

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 2.2.5.6. The required type signature and a sample test are:

```
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 are:

```
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 predicate is applied to the elements of the initial sequence, and values that pass the predicate are mapped. 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 at most n. The required type signature and a sample test:

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

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

scala> byLength(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 type signature and a 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]]`. Use the methods `updated` and `getOrElse` to work with dictionaries. The method `getOrElse` fetches a value from a dictionary by key but returns a default value if the key is not in the dictionary:

```
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 ("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 `digitsOf` using `map`, `zip`, or `foldLeft`, because these methods work only if we *already have* a sequence; but the function `digitsOf` needs to create a new sequence. We could create a sequence via the expression `(1 to n)` if the required length of the sequence were known in advance. However, the function `digitsOf` must produce a sequence whose length is determined by a condition that we cannot easily evaluate in advance.

A general "unfolding" operation needs to build a sequence whose length is not determined in advance. This kind of sequence is called a **stream**. The elements of a stream are computed only when necessary (unlike the elements of `List` or `Array`, which are all computed in advance). The unfolding operation will compute next elements on demand; 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 another type of sequence. In this way, we can generate a sequence of initially unknown length according to any given requirements.

The Scala library has a general stream-producing function `Stream.iterate`.[2] 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 elements until it runs out of memory and crashes.

Streams have methods such as `map`, `filter`, and `flatMap` similar to sequences. For instance, the method `drop` 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 digits of, 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

[2] In Scala 3, the `Stream` class is replaced by `LazyList`.

the corresponding sequence of last digits, n_k mod 10 (in this example: $5, 0, ...$). The sequence n_k is defined using mathematical induction:

- Base case: $n_0 = n$, where n is a 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 tabulate 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, a complete implementation for `digitsOf` 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
  }
```

We can shorten the code by using the syntax `(_ % 10)` instead of `{ nk => nk % 10 }`:

```
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 a close correspondence to a definition 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 general way of creating 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 and depend on previous elements. An example of this kind is computing the partial sums of a given sequence x_i, say $b_k = \sum_{i=0}^{k-1} x_i$. 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: $b_0 = 0$.
- Inductive 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 inductive 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 works on sequences, such as `map`, `filter`, or `flatten`. Let us show, as an illustration, how to implement the method `scanLeft` 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 is:

```
1 def scanLeft[A, B](xs: Seq[A])(b0: B)(next: (B, A) => B): Seq[B] = {
2   val init: (B, Seq[B]) = (b0, Seq(b0))
3   val (_, result) = xs.foldLeft(init) {
4     case ((b, seq), x) =>
5       val newB = next(b, x)
6       (newB, seq :+ newB)
7   }
8   result
9 }
```

To implement the (nameless) updater function for `foldLeft` in lines 4–6, we used a 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:

Definition by induction	Scala code example
$f([\,]) = b\ ;\ \ f(s{+\!\!+}[x]) = g(f(s), x)$	`f(xs) = xs.foldLeft(b)(g)`
$f([\,]) = b\ ;\ \ f([x]{+\!\!+}s) = g(x, f(s))$	`f(xs) = xs.foldRight(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.

the first is a tuple `(B, Seq[B])`, the second is a value of type `A`. Although the pattern expression `case ((b, seq), x) => ...` appears to match a nested tuple, it is just a special syntax. 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 number of ways for 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.
- Implement functions with type parameters.
- Use arbitrary inductive (i.e., recursive) formulas to:
  - convert sequences to single values (aggregation or "folding");
  - create new sequences from single values ("unfolding");
  - transform existing sequences into new sequences.

Table 2.1 shows Scala code implementing those tasks. Iterative calculations are implemented by translating mathematical induction directly into code. In the functional programming paradigm, the programmer does not need to write loops or use array indices. Instead, the programmer reasons about sequences as mathematical values: "Starting from this value, we get that sequence, then transform it into that other sequence," etc. This is a powerful way of working with sequences, dictionaries, and sets. Many kinds of programming errors (such as using an incorrect array index) are avoided from the outset, and the code is shorter and easier to read than code written via loops.

What problems cannot be solved with these tools? There is no automatic recipe for converting an arbitrary function into a tail-recursive one. 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* tail call.)

What if our recursive code cannot be transformed into tail-recursive code via the accumulator trick, but the recursion depth is so large that stack overflows occur? There exist special techniques (e.g., "continuations" and "trampolines") that convert non-tail-recursive code into code that runs without stack overflows. Those techniques are beyond the scope of this chapter.

2.5.1 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. Test with $f(x) = 2x + 1$.

Solution Define a stream of values $[1, f(1), f(f(1)), ...]$ and use `takeWhile` to stop the stream when the values reach 1000. The number n is then found as 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[Int]`, 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) { (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 always produced by `scanLeft`.

To test this function, let us define a stream whose values go up and down:

```
val s = Stream.iterate(0)(x => 1 - 2 * x)

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 data structure such as a priority queue that automatically keeps values sorted and its length bounded. For the purposes of this example, let us avoid using special 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]()) { (seq, x) =>
  // Sort in descending order, and take the first k values.
    (seq :+ x).sorted.reverse.take(k)
  }.drop(1) // Skip the undesired 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. (Hint: 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(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 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 { (x, y) => y }
```

Example 2.5.1.4 **(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 brown a fox")
res0: Map[String, Int] = Map(a -> 4, quick -> 2, brown -> 1, 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)
  }
}
```

An alternative, shorter implementation of the same function is:

```
def countWords(s: String): Map[String, Int] =
  s.split(" ").groupBy(w => w).map { case (w, xs) => (w, xs.length) }
```

The `groupBy` creates a dictionary in one function call rather than one entry at a time. But the resulting dictionary contains word lists instead of word counts, so we use `map` to compute the length of each word list:

```
scala> "a a b b b c".split(" ").groupBy(w => w)
res0: Map[String,Array[String]] = Map(b -> Array(b, b, b), a -> Array(a, a), c
    -> Array(c))

scala> res0.map { case (w, xs) => (w, xs.length) }
res1: Map[String,Int] = Map(b -> 3, a -> 2, c -> 1)
```

(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 (except for renaming `word` to `x`) and replace `String` by a type parameter `A` in the type signature:

```
def countValues[A](xs: Seq[A]): Map[A, Int] =
  xs.foldLeft(Map[A, Int]()) { (dict, x) =>
    val newCount = dict.getOrElse(x, 0) + 1
    dict.updated(x, newCount)
  }

scala> countValues(Seq(100, 100, 200, 100, 200, 200, 100))
res0: Map[Int,Int] = Map(100 -> 4, 200 -> 3)
```

Example 2.5.1.5 **(a)** Implement the binary search algorithm for a sorted sequence `xs: Seq[Int]` as a function returning the index of the requested value `goal` (assume that `xs` always contains `goal`):

```
@tailrec def binSearch(xs: Seq[Int], goal: Int): Int = ???

scala> binSearch(Seq(1, 3, 5, 7), 5)
res0: Int = 2
```

(b) Implement `binSearch` using `Stream.iterate` without explicit recursion.

Solution **(a)** The 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:

```
@tailrec def binSearch(xs: Seq[Int], goal: Int, acc: _ = ???): Int = {
  if (???) acc  // This condition must decide whether we are finished.
  else {
    // Determine which half of the sequence contains ‘goal‘.
    // Then update the accumulator accordingly.
    val newAcc = ???
    binSearch(xs, goal, newAcc) // Tail-recursive call.
  }
}
```

We will first decide the type and the initial value of the accumulator, then implement the updater.

The information required for the recursive call must show the segment of the sequence where the target number is present. That 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 $x_i \leq x_n \leq x_{j-1}$. 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 $i + 1 = j$. For convenience, we introduce *two* accumulator values (`left` and `right`) for i and j:

```
@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 set `xs.length` as a default value for the argument `right` of `binSearch`. This works because `right` is in a different **argument list** from `xs`. Default values in an argument list may depend on arguments in a *previous* argument list. However, this code:

```
def binSearch(xs: Seq[Int], goal: Int, left: Int = 0, right: Int = xs.length)
```

will generate an error. Arguments in the same argument list 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 stream 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 detect that, we use the `find` method:

```
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 || 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)
    .find { case (x, y) => y - x <= 1 } // Find an element with tight bounds.
    .get._1                             // Take the ‘left‘ bound from that.
}
```

In this code, recursion is delegated to `Stream.iterate` and is cleanly separated from the “business logic” (i.e., implementing the base case, the inductive step, and the post-processing).

Example 2.5.1.6 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` and `scanLeft`.

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 similarly to Section 2.3:

```
def SD(n: Int): Int = Stream.iterate(n)(_ / 10).takeWhile(_ != 0).map(_ %
    10).sum
```

Let us compute the sequence $[s_0, s_1, s_2, ...]$ by repeatedly applying `SD` to some number, say, `99`:

```
scala> Stream.iterate(99)(SD).take(10).toList
res1: List[Int] = List(99, 18, 9, 9, 9, 9, 9, 9, 9, 9)
```

We need to stop the stream when the values start to repeat, keeping the first repeated value. In the example above, we need to stop the stream after the value `9` (but include that value). One solution is to transform the stream via `scanLeft` into a stream of *pairs* of consecutive values, so that it becomes easier to detect repetition:

```
scala> Stream.iterate(99)(SD).scanLeft((0,0)) { case ((prev, x), next) => (x,
    next) }.take(8).toList
res2: List[(Int, Int)] = List((0,0), (0,99), (99,18), (18,9), (9,9), (9,9),
    (9,9), (9,9))

scala> res2.drop(1).takeWhile { case (x, y) => x  != y }
res3: 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(n)(SD)          // Stream[Int]
  .scanLeft((0,0)) { case ((prev, x), next) => (x, next) } // Stream[(Int, Int)]
  .drop(1).takeWhile { case (x, y) => x  != y }   // Stream[(Int, Int)]
  .map(_._2)                                      // Stream[Int]
  .toList                                         // List[Int]

scala> sdSeq(99)
res3: Seq[Int] = List(99, 18, 9)
```

Example 2.5.1.7 Implement a function `unfold` with the type signature:

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A]
```

The function should create a stream of values of type `A` with the initial value `init`. Next elements are computed from previous ones via the function `next` until it returns `None`. (The type `Option` is explained in Section 3.2.3.) An example test:

```
scala> unfold(0) { x => if (x > 5) None else Some(x + 2) }
res0: Stream[Int] = Stream(0, ?)

scala> res0.toList
res1: List[Int] = List(0, 2, 4, 6)
```

Solution We can formulate the task as an inductive definition of a stream. If `next(init) == None`, the stream will have just one value (`init`). This is the base case of the induction. Otherwise, `next(init) == Some(x)` yields a new value `x`. So, we need to continue to "unfold" the stream with `x` instead of `init`. (This is the inductive step.) To create streams with given values, we use the Scala library method `Stream.cons`. It constructs a stream from a head value and a tail stream:

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A] = next(init) match {
  case None      => Stream(init)    // A stream having a single value ‘init‘.
  case Some(x)   => Stream.cons(init, unfold(x)(next)) // ‘init‘ and then the
    tail of the stream.
}
```

Example 2.5.1.8 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 is defined as $h_{2k} = h_{2k+1} = s_k$ for $k = 0, 1, 2, ...$

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 2.5.1.9 (The **loop detection** problem.) 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, for an input sequence $s = [1, 3, 5, 7, 9, 3, 5, 7, 9, ...]$ we obtain the half-speed sequence $h = [1, 1, 3, 3, 5, 5, 7, 7, 9, 9, 3, 3, ...]$. Looking for an index $k > 0$ such that $h_k = s_k$, we find that $s_7 = h_7 = 7$. The element s_7 indeed 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 for use.

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 2.5.1.10 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`:

```
def revWords(s: String): String = s.split(" ").map(_.reverse).mkString(" ")
```

Example 2.5.1.11 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. A `filter` will then remove 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 2.5.1.12 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.

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 that extra information in our case?

Imagine creating the set of *all* subsequences that have no adjacent duplicates. For the input sequence $[1,2,2,5,4,4,4,8,2,3,3]$, this set of all subsequences will 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 $\{[1,2]\,,[2,5,4]\,,[4,8]\}$. When we see the next element, 2, we will update that 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‘ is not considered to 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 Define a function `dsq` that computes the sum of squared digits of a given integer; for example, `dsq(123) = 14` (see Example 2.5.1.6). Generalize `dsq` to take as an argument a function `f: Int => Int` replacing the squaring operation. The required type signature and a sample test:

```
def digitsFSum(x: Int)(f: Int => Int): Int = ???

scala> digitsFSum(123){ x => x * x }
res0: Int = 14

scala> digitsFSum(123){ x => x * x * x }
res1: Int = 36
```

Exercise 2.5.2.2 Compute the **Collatz sequence** c_i as a stream defined by:

$$c_0 = n \quad ; \qquad c_{k+1} = \begin{cases} c_k/2 & \text{if } c_k \text{ is even,} \\ 3 * c_k + 1 & \text{if } c_k \text{ is odd.} \end{cases}$$

Stop the stream when it reaches 1 (as one would expect[3] 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 = ???
```

[3] https://en.wikipedia.org/wiki/Collatz_conjecture

```
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: instead of just three sets `a`, `b`, `c`, a `Set[Set[Int]]` is given. 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 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") // Words are separated by a single
    space.
res0: String = "fox brown quick A"
```

Exercise 2.5.2.8 **(a)** Reverse an integer's digits (see Example 2.5.1.6) as shown:

```
def revDigits(n: Int): Int = ???

scala> revDigits(12345)
res0: Int = 54321
```

(b) A **palindrome integer** is an integer number `n` such that `revDigits(n) == n`. Write a predicate function of type `Int => Boolean` that checks whether a given positive integer is a palindrome.

Exercise 2.5.2.9 Define a function `findPalindrome: Long => Long` performing the following computation: First define `f(n) = revDigits(n) + n` for a given integer `n`, where the function `revDigits` was defined in Exercise 2.5.2.8. If `f(n)` is a palindrome integer, `findPalindrome` returns that integer. Otherwise, it keeps applying the same transformation and computes `f(n)`, `f(f(n))`, ..., until a palindrome integer is eventually found (this is mathematically guaranteed). A sample test:

```
scala> findPalindrome(10101)
res0: Long = 10101

scala> findPalindrome(123)
res0: Long = 444

scala> findPalindrome(83951)
res1: Long = 869363968
```

Exercise 2.5.2.10 Transform a given sequence `xs: Seq[Int]` into a sequence of type `Seq[(Int, Int)]` of pairs that skip one neighbor. Implement this transformation as a function `skip1` with a type parameter `A` instead of the type `Int`. The required type signature and a sample test:

```
def skip1[A](xs: Seq[A]): Seq[(A, A)] = ???

scala> skip1(List(1, 2, 3, 4, 5))
res0: List[Int] = List((1,3), (2,4), (3,5))
```

Exercise 2.5.2.11 **(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 (integers k such that $1 \leq k < n$ and k divides n). 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 2.5.2.12 Transform a sequence by removing adjacent repeated elements 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 2.5.2.13 Implement a function `unfold2` with the type signature:

```
def unfold2[A, B](init: A)(next: A => Option[(A, B)]): Stream[B]
```

The function should create a stream of values of type `B` by repeatedly applying the given function `next` until it returns `None`. At each iteration, `next` should be applied to the value of type `A` returned by the previous call to `next`. An example test:

```
scala> unfold2(0) { x => if (x > 5) None else Some((x + 2, s"had $x")) }
res0: Stream[String] = Stream(had 0, ?)

scala> res0.toList
res1: List[String] = List(had 0, had 2, had 4)
```

Exercise 2.5.2.14 **(a)** Remove repeated elements (whether adjacent or not) from a sequence of type `Seq[A]`. (This reproduces the standard library's 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 2.5.2.15 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 techniques and `foldLeft`.

Exercise 2.5.2.16 Using tail recursion, find all common integers between two *sorted* sequences:

```
@tailrec def commonInt(xs: Seq[Int], ys: Seq[Int]): Seq[Int] = ???

scala> commonInt(Seq(1, 3, 5, 7), Seq(2, 3, 4, 6, 7, 8))
res0: Seq[Int] = List(3, 7)
```

2.6 Discussion and further developments

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.