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.

Draft version of March 23, 2021

Published by **lulu.com** in 2021

Printed copies may be ordered at http://www.lulu.com/content/paperback-book/24915714

ISBN: 978-0-359-76877-6

Source hash (sha256): 8725d4abf71910c704bf5a279acb4a622c159cdc259af2bf0f50d2833b71ce66

Git commit: 6efa5c4c818d5218602dff90edcf0681a7b15c03

PDF file built by pdfTeX 3.14159265-2.6-1.40.20 (TeX Live 2019) on Tue, 23 Mar 2021 21:25:22 +0100 by Darwin 19.6.0

A *Transparent* copy of the source code for the book is available at https://github.com/winitzki/sofp/releases 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" as `sofp.pdf` or `sofp-draft.pdf`. The source code may be also included as a "file attachment" named `sofp-src.tar.bz2` within a PDF file. To extract, run the command '`pdftk sofp.pdf unpack_files output .`' and then '`tar jxvf sofp-src.tar.bz2`'. See the file `README.md` for compilation instructions.

This book is a pedagogical in-depth tutorial and reference on the theory of functional programming (FP) as practiced in the early 21st 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 collections; recursive functions and types; the Curry-Howard correspondence; laws, structural analysis, and code for functors, monads, and other typeclasses; techniques of symbolic derivation and proof; parametricity theorems; and free type constructions.

Long and difficult, yet boring explanations are logically developed in excruciating detail through 1480 Scala code snippets, 151 statements with step-by-step derivations, 97 diagrams, 198 solved examples with tested Scala code, and 223 exercises. Discussions further build upon each chapter's material.

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 Yoneda identities for reasoning about the Church encoding and the free type constructions.

Readers should have a working knowledge of programming; e.g., be able to write code that prints the number of distinct words in a small text file. The difficulty of this book's mathematical derivations is at the level of high-school calculus, similar to that of multiplying matrices and simplifying the expressions

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

Sergei Winitzki received a Ph.D. in theoretical physics. After a career in academic research, he currently works as a software engineer.

Contents

Preface 1
Formatting conventions used in this book . . . 2

I Beginner level 3

1 Mathematical formulas as code. I. Nameless functions 4
1.1 Translating mathematics into code . . . 4
1.1.1 First examples . . . 4
1.1.2 Nameless functions . . . 5
1.1.3 Nameless functions and bound variables . . . 7
1.2 Aggregating data from sequences . . . 8
1.3 Filtering and truncating a sequence . . . 10
1.4 Solved examples . . . 11
1.4.1 Aggregation . . . 11
1.4.2 Transformation . . . 12
1.5 Summary . . . 13
1.6 Exercises . . . 14
1.6.1 Aggregation . . . 14
1.6.2 Transformation . . . 15
1.7 Discussion . . . 15
1.7.1 Functional programming as a paradigm . . . 15
1.7.2 The mathematical meaning of “variables” . . . 16
1.7.3 Iteration without loops . . . 17
1.7.4 Nameless functions in mathematical notation . . . 17
1.7.5 Named and nameless expressions and their uses . . . 19
1.7.6 Historical perspective on nameless functions . . . 20

2 Mathematical formulas as code. II. Mathematical induction 21
2.1 Tuple types . . . 21
2.1.1 Examples of using tuples . . . 21
2.1.2 Pattern matching for tuples . . . 22
2.1.3 Using tuples with collections . . . 23
2.1.4 Treating dictionaries as collections . . . 24
2.1.5 Solved examples: Tuples and collections . . . 27
2.1.6 Reasoning about type parameters in collections . . . 31
2.1.7 Exercises: Tuples and collections . . . 32
2.2 Converting a sequence into a single value . . . 33
2.2.1 Inductive definitions of aggregation functions . . . 34
2.2.2 Implementing functions by recursion . . . 35
2.2.3 Tail recursion . . . 35
2.2.4 Implementing general aggregation (`foldLeft`) . . . 39
2.2.5 Solved examples: using `foldLeft` . . . 41
2.2.6 Exercises: Using `foldLeft` . . . 44
2.3 Converting a single value into a sequence . . . 45

2.4 Transforming a sequence into another sequence . . . 47
2.5 Summary . . . 47
2.5.1 Solved examples . . . 48
2.5.2 Exercises . . . 55
2.6 Discussion and further developments . . . 58
2.6.1 Total and partial functions . . . 58
2.6.2 Scope and shadowing of pattern matching variables . . . 59
2.6.3 Lazy values and sequences. Iterators and streams . . . 59

3 The logic of types. I. Disjunctive types 63
3.1 Scala's case classes . . . 63
3.1.1 Tuple types with names . . . 63
3.1.2 Case classes with type parameters . . . 65
3.1.3 Tuples with one part and with zero parts . . . 66
3.1.4 Pattern matching for case classes . . . 67
3.2 Disjunctive types . . . 67
3.2.1 Motivation and first examples . . . 67
3.2.2 Solved examples: Pattern matching for disjunctive types . . . 69
3.2.3 Standard disjunctive types: `Option`, `Either`, `Try` . . . 72
3.3 Lists and trees: recursive disjunctive types . . . 78
3.3.1 Lists . . . 78
3.3.2 Tail recursion with `List` . . . 80
3.3.3 Binary trees . . . 83
3.3.4 Rose trees . . . 83
3.3.5 Regular-shaped trees . . . 84
3.3.6 Abstract syntax trees . . . 86
3.4 Summary . . . 88
3.4.1 Solved examples . . . 88
3.4.2 Exercises . . . 91
3.5 Discussion and further developments . . . 92
3.5.1 Disjunctive types as mathematical sets . . . 92
3.5.2 Disjunctive types in other programming languages . . . 94
3.5.3 Disjunctions and conjunctions in formal logic . . . 95

II Intermediate level 96

4 The logic of types. II. Curried functions 97
4.1 Functions that return functions . . . 97
4.1.1 Motivation and first examples . . . 97
4.1.2 Curried and uncurried functions . . . 98
4.1.3 Equivalence of curried and uncurried functions . . . 99
4.2 Fully parametric functions . . . 100
4.2.1 Examples. Function composition . . . 101
4.2.2 Laws of function composition . . . 103
4.2.3 Example: A function that is *not* fully parametric . . . 105
4.3 Symbolic calculations with nameless functions . . . 106
4.3.1 Calculations with curried functions . . . 106
4.3.2 Solved examples: Deriving a function's type from its code . . . 108
4.4 Summary . . . 111
4.4.1 Solved examples . . . 111
4.4.2 Exercises . . . 116

4.5 Discussion and further developments . . . 117
4.5.1 Higher-order functions . . . 117
4.5.2 Name shadowing and the scope of bound variables . . . 118
4.5.3 Operator syntax for function applications . . . 118
4.5.4 Deriving a function's code from its type . . . 120

5 The logic of types. III. The Curry-Howard correspondence 121
5.1 Values computed by fully parametric functions . . . 121
5.1.1 Motivation . . . 121
5.1.2 Type notation and $\mathcal{CH}$-propositions for standard type constructions . . . 122
5.1.3 Solved examples: Type notation . . . 125
5.1.4 Exercises: Type notation . . . 128
5.2 The logic of $\mathcal{CH}$-propositions . . . 128
5.2.1 Motivation and first examples . . . 128
5.2.2 Example: Failure of Boolean logic when reasoning about $\mathcal{CH}$-propositions . . . 130
5.2.3 The rules of proof for $\mathcal{CH}$-propositions . . . 131
5.2.4 Example: Proving a $\mathcal{CH}$-proposition and deriving code . . . 135
5.3 Solved examples: Equivalence of types . . . 138
5.3.1 Logical identity does not correspond to type equivalence . . . 138
5.3.2 Arithmetic identity corresponds to type equivalence . . . 141
5.3.3 Type cardinalities and type equivalence . . . 145
5.3.4 Type equivalence involving function types . . . 147
5.4 Summary . . . 153
5.4.1 Solved examples . . . 154
5.4.2 Exercises . . . 163
5.5 Discussion and further developments . . . 163
5.5.1 Using the Curry-Howard correspondence for writing code . . . 163
5.5.2 Implications for designing new programming languages . . . 165
5.5.3 Uses of the void type (`Nothing`) . . . 166
5.5.4 Relationship between Boolean logic and constructive logic . . . 167

6 Functors and contrafunctors 170
6.1 Practical use . . . 170
6.1.1 Motivation. Type constructors that wrap data . . . 170
6.1.2 Extended example: `Option` and the identity law . . . 171
6.1.3 Motivation for the composition law . . . 172
6.1.4 Functors: definition and examples . . . 173
6.1.5 Functor block expressions . . . 177
6.1.6 Examples of non-functors . . . 180
6.1.7 Contrafunctors . . . 185
6.1.8 Covariance, contravariance, and subtyping . . . 187
6.1.9 Solved examples: implementation of functors and contrafunctors . . . 189
6.1.10 Exercises: implementation of functors and contrafunctors . . . 193
6.2 Laws and structure . . . 194
6.2.1 Reformulations of laws . . . 194
6.2.2 Bifunctors . . . 195
6.2.3 Constructions of functors . . . 197
6.2.4 Constructions of contrafunctors . . . 204
6.2.5 Solved examples: How to recognize functors and contrafunctors . . . 206
6.3 Summary . . . 209
6.3.1 Exercises: Functor and contrafunctor constructions . . . 209

6.4 Further developments . . . 210
6.4.1 Profunctors . . . 210
6.4.2 Subtyping with injective or surjective conversion functions . . . 211

7 Reasoning about code. Techniques of symbolic derivation 213
7.1 Mathematical code notation . . . 213
7.1.1 The nine constructions of fully parametric code . . . 213
7.1.2 Function composition and the pipe notation . . . 216
7.1.3 Functor and contrafunctor liftings . . . 217
7.2 Derivation techniques . . . 218
7.2.1 Auxiliary functions for handling products . . . 218
7.2.2 Deriving laws for functions with known implementations . . . 218
7.2.3 Working with disjunctive types in matrix notation . . . 219
7.2.4 Derivations involving unknown functions with laws . . . 222
7.2.5 Exercises . . . 224

8 Typeclasses and functions of types 225
8.1 Motivation and first examples . . . 225
8.1.1 Constraining type parameters . . . 225
8.1.2 Functions of types and values . . . 225
8.1.3 Partial functions of types and values . . . 226
8.2 Implementing typeclasses . . . 227
8.2.1 Creating a partial function at type level . . . 227
8.2.2 Scala's `implicit` values . . . 229
8.2.3 Implementing typeclasses by making instances `implicit` . . . 230
8.2.4 Extension methods . . . 231
8.2.5 Solved examples: Implementing typeclasses in practice . . . 232
8.2.6 Typeclasses for type constructors . . . 237
8.3 Deriving typeclass instances via structural analysis of types . . . 239
8.3.1 Extractors . . . 239
8.3.2 Equality comparison: The `Eq` typeclass . . . 244
8.3.3 Semigroups . . . 247
8.3.4 Monoids . . . 251
8.3.5 Pointed functors: motivation and laws . . . 255
8.3.6 Pointed functors: structural analysis . . . 257
8.3.7 Co-pointed functors . . . 260
8.3.8 Pointed contrafunctors . . . 264
8.4 Summary . . . 266
8.4.1 Solved examples . . . 267
8.4.2 Exercises . . . 274
8.5 Further developments . . . 275
8.5.1 The existence of values for recursive types . . . 275
8.5.2 Proofs of associativity of `concat` for lists and arrays . . . 278
8.5.3 Higher-order type functions. Kinds . . . 280
8.5.4 Inductive typeclasses and their properties . . . 281
8.5.5 Typeclasses with several type parameters (type relations) . . . 284
8.5.6 Inheritance and automatic conversions of typeclasses . . . 285

9 Computations in functor blocks. I. Filterable functors and contrafunctors 288
9.1 Practical uses of filtering . . . 288
9.1.1 Examples and intuitions for the filtering operation . . . 289
9.1.2 The laws of filtering: Motivation and derivation . . . 290
9.1.3 Examples of non-filterable functors . . . 292

9.1.4 Solved examples: Programming with filterable functors . . . 293
9.1.5 Exercises: Programming with filterable functors . . . 298
9.2 Laws and structure . . . 299
9.2.1 Simplifying the filtering laws: Motivation for `deflate` . . . 299
9.2.2 Equivalence of `filter` and `deflate` . . . 301
9.2.3 Motivation and laws for `liftOpt` . . . 306
9.2.4 Constructions of filterable functors . . . 311
9.2.5 Filterable contrafunctors: motivation and examples . . . 321
9.2.6 Constructions of filterable contrafunctors . . . 324
9.3 Summary . . . 328
9.3.1 Solved examples . . . 328
9.3.2 Exercises . . . 333
9.4 Further developments . . . 334
9.4.1 Naturality laws and natural transformations . . . 334
9.4.2 Generalizing the laws of liftings. Kleisli functions . . . 337
9.4.3 Motivation for using category theory . . . 338

10 Computations in functor blocks. II. Semimonads and monads 342
10.1 Practical use of monads . . . 342
10.1.1 Motivation for semimonads: Nested iteration . . . 342
10.1.2 List-like monads . . . 344
10.1.3 Pass/fail monads . . . 352
10.1.4 Tree-like semimonads and monads . . . 354
10.1.5 The `Reader` monad . . . 358
10.1.6 The `Writer` monad . . . 361
10.1.7 The `State` monad . . . 362
10.1.8 The eager/lazy evaluation monad (`Eval`) . . . 363
10.1.9 The continuation monad (`Cont`) . . . 364
10.1.10 Exercises . . . 367
10.2 Laws of semimonads and monads . . . 368
10.2.1 Motivation for the semimonad laws . . . 368
10.2.2 The laws of `flatten` . . . 369
10.2.3 Verifying the associativity law via `flatten` . . . 371
10.2.4 From semimonads to monads: Motivation for the identity laws . . . 375
10.2.5 The monad identity laws in terms of `pure` and `flatten` . . . 376
10.2.6 Monad laws in terms of Kleisli functions . . . 376
10.2.7 Verifying the monad laws using Kleisli functions . . . 379
10.2.8 Structural analysis of semimonads and monads . . . 380
10.2.9 Exercises: laws and structure of monads . . . 393
10.3 Further developments . . . 394
10.3.1 Why monads must be functors . . . 394
10.3.2 Equivalence of a natural transformation and a “lifting” . . . 396
10.3.3 Monads, effects, and runners . . . 397
10.3.4 Monads in category theory. Monad morphisms . . . 399
10.3.5 Constructions of polynomial monads . . . 403
10.3.6 Constructions of M-filterable functors and contrafunctors . . . 403

11 Applicative functors and contrafunctors 405
11.1 Motivation: monadic programs with independent effects . . . 405
11.2 Slides, Part I . . . 405
11.2.1 Exercises I . . . 406

11.3 Slides, Part II . . . 406
11.3.1 Exercises . . . 414
11.4 Practical use . . . 415
11.5 Laws and structure . . . 415

12 Traversable functors 416
12.1 Slides . . . 416
12.1.1 Monoid morphisms . . . 419
12.1.2 Exercises . . . 419
12.2 Discussion . . . 420

III Advanced level 421

13 "Free" type constructions 422
13.1 Slides . . . 422
13.1.1 Exercises . . . 433
13.2 Properties of free constructions . . . 434
13.2.1 Free monad . . . 434
13.3 Working with quantified types . . . 437
13.3.1 The Yoneda identities . . . 437
13.3.2 Recursive type equations with different fixpoints . . . 439
13.3.3 The Church encoding of recursive types . . . 440
13.3.4 The co-Yoneda identities . . . 444
13.4 Discussion . . . 444
13.4.1 Beyond Yoneda: using parametricity to simplify quantified types . . . 444

14 Computations in functor blocks. III. Monad transformers 446
14.1 Practical use . . . 446
14.1.1 Combining monadic effects via functor composition . . . 446
14.1.2 Combining monads via monad transformers . . . 449
14.1.3 Monad transformers for standard monads . . . 451
14.1.4 Combining more than two monads: monad stacks . . . 457
14.1.5 A typeclass for monad transformers . . . 461
14.1.6 Lifts and runners for monad stacks . . . 461
14.1.7 Constructing lifts via type relations . . . 463
14.1.8 Constructing lifts via typeclasses for monad operations ("MTL-style") . . . 464
14.2 Laws of monad transformers . . . 468
14.2.1 Motivation for the laws of lifts . . . 468
14.2.2 Motivation for the laws of runners . . . 470
14.2.3 Simplifying the laws of lifts and runners via category theory . . . 472
14.2.4 Summary of the laws of monad transformers . . . 475
14.2.5 Examples of invalid monad transformers . . . 476
14.2.6 Examples of failure to define a generic monad transformer . . . 476
14.2.7 Functor composition with transformed monads . . . 479
14.2.8 Stacking two monads. Proofs . . . 479
14.2.9 Stacking any number of monads. Proofs . . . 482
14.3 Common properties of compositional transformers . . . 482
14.3.1 Motivation for the `swap` function . . . 483
14.3.2 Deriving the necessary laws for `swap` . . . 484
14.3.3 Deriving `swap` from `flatten` . . . 487
14.3.4 Monad transformer identity law. Proofs . . . 489
14.3.5 Monad transformer lifting laws. Proofs . . . 490

14.3.6 Monad transformer runner laws. Proofs . . . 491
14.3.7 Summary of results . . . 495
14.4 Composed-inside transformers. Linear monads . . . 495
14.4.1 Definitions of `swap` and `flatten` . . . 496
14.4.2 The laws of `swap`. Proofs . . . 496
14.4.3 Composition of transformers for linear monads . . . 501
14.5 Composed-outside transformers. Rigid monads . . . 501
14.5.1 Rigid monad construction 1 (product) . . . 502
14.5.2 Rigid monad construction 2 (composition) . . . 502
14.5.3 Rigid monad construction 3 (choice) . . . 503
14.6 Transformers for some monad constructions . . . 512
14.6.1 Transformer for products of monads. Proofs . . . 512
14.6.2 Transformer for free pointed monads. Proofs . . . 513
14.7 Transformers for recursive monads . . . 524
14.7.1 Transformer for free monads. Proofs . . . 524
14.7.2 Transformer for the `List` monad. Proofs . . . 536
14.8 Incomplete transformers . . . 550
14.8.1 Transformer for the state monad. Proofs . . . 550
14.8.2 Transformer for the continuation monad. Proofs . . . 554
14.8.3 Transformer for the generalized `Search` monad. Proofs . . . 555
14.8.4 The codensity monad and its transformer. Proofs . . . 558
14.8.5 Examples of monads with two different transformers . . . 562
14.9 Further developments . . . 563
14.9.1 Some properties of monad morphisms . . . 563
14.9.2 Does a composition of rigid monads have a `swap` function? . . . 564
14.9.3 Rigid functors and their properties . . . 568
14.9.4 Monadically natural lifting of MTL operations (Jaskelioff's "`Monatron`") . . . 575
14.9.5 Exercises . . . 589

IV Discussions **593**

15 Summary and problems **594**
15.1 Exercises in applied functional type theory . . . 594
15.1.1 Exercises . . . 594
15.1.2 Open problems . . . 595

16 "Applied functional type theory": A proposal **598**
16.1 AFTT is not covered by courses in computer science . . . 598
16.2 AFTT is not category theory, type theory, or formal logic . . . 599

17 Essay: Software engineers and software artisans **603**
17.1 Engineering disciplines . . . 603
17.2 Artisanship: Trades and crafts . . . 603
17.3 Programmers today are artisans, not engineers . . . 604
17.3.1 No requirement of formal study . . . 604
17.3.2 No mathematical formalism guides software development . . . 605
17.3.3 Programmers avoid academic terminology . . . 606
17.4 Towards true engineering in software . . . 606
17.5 Does software need engineers, or are artisans good enough? . . . 608

18 Essay: Towards functional data engineering with Scala **609**
18.1 Data is math . . . 609

18.2 Functional programming is math . . . 609
18.3 The power of abstraction . . . 610
18.4 Scala is Java on math . . . 611
18.5 Summary . . . 611

19 Essay: Why category theory is useful in functional programming 612
19.1 A "types/functions" category for a programming language . . . 612
19.2 The use of endofunctors . . . 613
19.3 The use of natural transformations . . . 613
19.4 Other properties of the "types/functions" category . . . 614
19.5 Some useful sub-categories of endofunctors . . . 614
19.5.1 Filterable endofunctors . . . 615
19.5.2 Monadic endofunctors . . . 615
19.6 Category theory and the laws of FP idioms . . . 616

V Appendixes 617

A Notations 618
A.1 Summary of notations for types and code . . . 618
A.2 Detailed explanations . . . 619

B Glossary of terms 624
B.1 How the term "algebra" is used in functional programming . . . 626

C Inferring code from types with the LJT algorithm 627
C.1 Slides . . . 627

D Parametricity theorem and naturality laws 632
D.1 Commutativity laws for profunctors and bifunctors . . . 633
D.1.1 Proof of the profunctor commutativity law . . . 634
D.1.2 Commutativity laws for bifunctors and bi-contrafunctors . . . 636
D.2 Naturality laws for fully parametric functions . . . 636
D.2.1 Dinatural transformations between profunctors . . . 637
D.2.2 Composition properties of natural and dinatural transformations . . . 639
D.2.3 Proof of the parametricity theorem via dinaturality . . . 641
D.2.4 Uniqueness of functor and contrafunctor typeclass instances . . . 647
D.3 Parametricity theorem formulated via relations . . . 648
D.3.1 Relations between types. Relational lifting . . . 649
D.3.2 Relational parametricity theorem. Proofs . . . 662
D.3.3 Strong dinaturality: definition and general properties . . . 665
D.3.4 Strong dinaturality derived from parametricity . . . 674
D.4 Summary . . . 678

E Solutions of some exercises 680

F A humorous disclaimer 697

G GNU Free Documentation License 698
G.0.0 Applicability and definitions . . . 698
G.0.1 Verbatim copying . . . 698
G.0.2 Copying in quantity . . . 698
G.0.3 Modifications . . . 698

List of Tables 700

List of Figures 701

Index 702

Preface

This book is at once 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 to advanced. The book assumes a certain amount of mathematical experience, at the level of familiarity with high-school algebra or calculus.

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. Only then the laws are formulated as mathematical equations and rigorously proved.

To achieve a clearer presentation of the material, the book uses some non-standard notations (Appendix A) and terminology (Appendix B). The presentation is self-contained, defining and explaining all required ideas, notations, and Scala language features.

Each concept and technique is motivated and explained to make it as simple as possible and also clarified via solved examples and exercises, which the readers will be able to solve after absorbing the preceding material. More difficult examples and exercises are marked by an asterisk (*).

A software engineer needs to know only a few fragments of mathematical theory; namely, the fragments that answer questions arising in the practice of functional programming. So, this book keeps theoretical material at the minimum; *ars longa, vita brevis*. (Chapter 16 discusses the scope of the required theory.) Mathematical generalizations are not pursued beyond proven practical relevance or immediate pedagogical usefulness. This reduces the required mathematical knowledge to first notions of category theory, type theory, and formal logic. Concepts such as functors or natural transformations arise organically from the practice of reasoning about code and are introduced without reference to category theory. This book does not use "introduction/elimination rules", "strong normalization", "complete partial orders", "adjoint functors", "pullbacks", or "topoi", because learning these concepts will not help a programmer write code.

This book is also *not* an introduction to current theoretical research in functional programming. Instead, the focus is on material known to be practically useful. That includes constructions such as the "filterable functor" and "applicative contrafunctor" but excludes a number of theoretical developments that do not (yet?) appear to have significant applications.

The first part of the book is for beginners in functional programming. Readers already familiar with functional programming could skim the glossary (Appendix B) to see the unfamiliar terminology and then start the book with 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 could benefit from working through Chapter 7, which summarizes the code notation more systematically and clarifies it with additional examples.

All code examples are intended only for explanation and illustration. As a rule, the code is not optimized for performance or stack safety.

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 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.

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 sequence of values to a *single* value.

- Equations are numbered per chapter: Eq. (1.3). Statements, examples, and exercises are numbered per subsection: Example 1.4.1.6 is in subsection 1.4.1, which belongs to Chapter 1.

- Scala code is written inline using a small monospaced font, such as `flatMap` or `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 new notation for types where, e.g., the Scala type expression `((A, B)) => Option[A]` is written as $A \times B \rightarrow \mathbb{1} + A$. Also, a new notation for code is introduced and developed in Chapters 5–7 for efficient reasoning about typeclass laws. For example, the functor composition law is written in the code notation as

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

  where L is a functor and $f^{:A \rightarrow B}$ and $g^{:B \rightarrow C}$ 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 $⨾$ denotes the forward composition of functions (Scala's `andThen` method). Appendix A summarizes the notation conventions for types and code.

- Frequently used methods of standard typeclasses, such as `pure`, `flatMap`, `flatten`, `filter`, etc., are denoted by shorter words and are labeled by the type constructor they belong to. For instance, the text talks about typeclass methods `pure`, `flatten`, and `flatMap` for a monad M but denotes the same methods by pu_M, ftn_M, and flm_M when writing code formulas.

- Derivations are written in a two-column format where the right column contains formulas in the code notation and the left column gives a line-by-line explanation or indicates the property or law used to derive the expression at right. A green underline in an expression shows the parts to be rewritten using the law or equation indicated in the *next* line:

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

  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 remember the steps of a derivation, the green underlines *play no role* in symbolic calculations.

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

Part I

Beginner level

1 Mathematical formulas as code. I. Nameless functions

1.1 Translating mathematics into code

1.1.1 First examples

We begin by implementing some computational tasks in Scala.

Example 1.1.1.1: Factorial of 10 Find the product of integers from 1 to 10 (the **factorial** of 10).

First, we write a mathematical formula for the result:

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

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

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

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

```
scala> 100 + (1 to 10).product + 100   // This code contains ‘(1 to 10).product‘ as 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: Factorial as a function Define a function that takes an integer n and computes the factorial of n.

A mathematical formula for this function can be written as

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

The corresponding Scala code is

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

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

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

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 `Int` in the Scala code. So, the argument's type in the code specifies the **domain** of a function (the set of admissible arguments).

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

```
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 $\rightarrow$ (pronounced "maps to") like this:

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

So the mathematical notation for the nameless factorial function is

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

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

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

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

```
n: Int
```

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

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

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

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

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

We see that the value `fac` has the type `Int => Int`, which means that the function `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 Scala interpreter prints `<function1>` as the "value" of `fac`. An alternative Scala interpreter, `ammonite`[2], prints something like this:

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

The long number could indicate an address in memory.

One may imagine that a "function value" represents a block of compiled code that will actually run and evaluate the function's body when the function is applied to its argument.

[1] In mathematics, the "maps to" symbol is $\mapsto$, but this book uses a simpler arrow symbol $\rightarrow$ that is visually similar. Many programming languages use the symbols `->` or `=>` for the function arrow; see Table 1.2.

[2] https://ammonite.io/

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A simple mathematical formula for this function can be written as

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

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

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

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

We can now apply the function `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 type of named functions. The required syntax looks like this:

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

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

1.1.3 Nameless functions and bound variables

The code for `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 involve any functions.

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` we have seen 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 mathematical formula

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

does not seem to use any functions 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 actually defined *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 defined only in the right-hand side, where it is first mentioned as the arbitrary element $k \in [2, n-1]$ and then used in the sub-expression "$n\%k$".

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 $n \neq 0 \bmod k$, which can be viewed as a certain given *function of* k that returns a `Boolean` value. Translating the mathematical notation into code, it is therefore 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 only within that function's body and cannot be used in any code outside the expression `n % k != 0`.

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

The main difference between free and bound variables is that bound variables can be *locally renamed* at will, unlike free variables. To see this, consider that we could rename k to z and write instead of Eq. (1.2) an equivalent definition,

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

or in Scala code:

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

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

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

As an example, the mathematical formula $\forall k \in [1, n]\,.\,p(k)$ 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 $k \rightarrow p(k)$ does exactly the same thing as the (named) function p: It takes an argument, which we may call k, and returns $p(k)$. So, we can simplify the Scala code above to

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

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

1.2 Aggregating data from sequences

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

In this example, the argument of `map` is the nameless function $x \to x^2 + 100x$. This function will be used repeatedly by `map` to transform each integer from `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
res3: List[Int] = List(101, 204, 309)
```

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

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

To test this code, let us run it in the Scala interpreter. In order to let the interpreter work correctly with code entered line by line, the dot character needs to be at the *end* of the line. (In compiled code, the dots may be at the beginning of line since the compiler reads the entire code at once.)

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

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

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

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

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

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

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

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

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

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

1.4.1 Aggregation

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

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

Solution: Use `takeWhile` to truncate the initial list when $k^2 \leq n$ 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: $\prod_{k\in[1,10]} |\sin(k+2)|$.

Solution

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

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

Solution

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

It is safe to compute $\sqrt{\cos k}$, because we have first filtered the list by keeping only values k for which $\cos k > 0$. 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[4] truncated up to $\frac{2n}{2n+1}$:

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

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

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)/(2*i + 1)

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

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

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

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

Example 1.4.1.6 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.

Solution

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

scala> euler_series(100000)
res0: Double = 1.6449240668982423

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

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

Example 1.4.1.7 Check numerically the infinite product formula

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

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

Solution

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

Here is a test for this function:

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

1.4.2 Transformation

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

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

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 the type of `t` from the code; no other type would work since 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 $1 \le j \le k$, each j satisfying the condition $j^2 > 2k$.

Solution

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

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

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

(shown at left) computes the list of j's that satisfy the condition $j^2 > 2k$, and then compares the size of that list 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 shows how to implement some often used mathematical constructions in Scala.

What problems can one solve with this knowledge?

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

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

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

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

where the given function f is applied n times.

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

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

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

These computations involve a general case of *mathematical induction*.

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 implement that in Scala code.

Example 2 can be formulated as a definition of a new list r by induction: the base case is $r_0 = s_0$, and the inductive step is $r_i = s_i + r_{i-1}$ for $i = 1, 2, 3, ...$ However, operations such as `map` and `filter` cannot compute r_i depending on the value of r_{i-1}.

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

Chapter 2 explains how to implement these tasks using recursion.

1.6 Exercises

1.6.1 Aggregation

Exercise 1.6.1.1 Machin's formula[5] converges to π faster than Example 1.4.1.5:

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

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

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

Exercise 1.6.1.2 Using the function `isPrime`, check numerically the Euler product formula[6] for the Riemann's zeta function $\zeta(4)$. It is known[7] that $\zeta(4) = \frac{\pi^4}{90}$:

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

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

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

[7] https://tinyurl.com/yxey4tsd

1.6.2 Transformation

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 j such that $2 \leq j < n$. Compute the set of all "3-factor" integers n among $n \in [1, ..., 1000]$.

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

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

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

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

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

1.7 Discussion

1.7.1 Functional programming as a paradigm

Functional programming (FP) is a **paradigm** of programming — 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 approach allows programmers to derive code through logical reasoning rather than through guessing, similarly to how books on mathematics reason about mathematical formulas and derive results systematically, without guessing or "debugging." 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, functional programming 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 easily 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`, etc.) that proved to be especially useful in real-life programming, although many of them are not standard in mathematical literature.

Mastering FP involves practicing to reason about programs as formulas "translated into code", building up the specific kind of applied mathematical intuition, and getting familiar with mathematical concepts adapted to a programmer's needs. The FP community has discovered a number of specific programing 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 productively used in any programming language that supports nameless functions.

1.7.2 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
```

or we could 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"
                    ^
```

1.7.3 Iteration without loops

A distinctive feature of the FP paradigm is handling of iteration without writing loops.

Iterative computations are ubiquitous in mathematics. As an example, consider the formula for the standard deviation (σ) estimated from a data sample $[x_1, ..., x_n]$:

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

This expression is computed by iterating over values of the index i. And yet, no mathematics textbook uses loops or says “now repeat this formula ten times”. Indeed, it would be pointless to evaluate a formula such as $x^2 + x$ ten times, because the result of $x^2 + x$ remains the same every time. It is also meaningless to “repeat” an equation such as $(x-1)\,(x^2+x+1) = x^3 - 1$.

Instead of loops, mathematicians write *expressions* such as $\sum_{i=1}^{n} s_i$, where symbols such as $\sum_{i=1}^{n}$ denote the results of entire iterative computations. Such computations are defined using mathematical induction. The FP paradigm has developed rich tools for translating mathematical induction into code. This chapter focused on methods such as `map`, `filter`, and `sum`, which implement certain kinds of iterative computations. These and other similar methods can be combined in flexible ways, enabling programmers to write iterative code without loops. For example, the computation of σ according to the formula shown above may be implemented 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.

1.7.4 Nameless functions in mathematical notation

Functions in mathematics are mappings from one set to another. A function does not necessarily *need* a name; the mapping just needs to be defined. However, nameless functions have not been widely used in the conventional mathematical notation. It turns out that nameless functions are 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

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

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

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

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

Now consider the traditional mathematical notations for summation, for instance,

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

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

$$\text{denote summation by }\sum_0^x\left(k\rightarrow\frac{1}{1+k}\right)\text{ instead of }\sum_{k=0}^{x}\frac{1}{1+k}\quad,$$

$$\text{denote integration by }\int_0^x\left(z\rightarrow\frac{1}{1+z}\right)\text{ instead of }\int_0^x\frac{dz}{1+z}\quad.$$

In this notation, the new summation symbol $\sum_0^x$ does not mention the name "k" but takes a function as an argument. Similarly, the new integration symbol $\int_0^x$ does not mention "z" and does not use the special symbol "d" but now takes a function as an argument. Written in this way, the operations of summation and integration become *functions* that take functions as arguments. The above summation may be written in a consistent and straightforward manner 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[8] is an algorithm for approximate numerical integration, defined by the formulas

$$\text{integration}\,(a,b,g,\varepsilon)=\frac{\delta}{3}\left(g(a)+g(b)+4s_1+2s_2\right)\quad,$$

$$\text{where}\quad n=2\left\lfloor\frac{b-a}{\varepsilon}\right\rfloor,\quad\delta_x=\frac{b-a}{n}\quad,$$

$$s_1=\sum_{k=1,3,...,n-1}g(a+k\delta_x)\quad,\qquad s_2=\sum_{k=2,4,...,n-2}g(a+k\delta_x)\quad.$$

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

Here is a straightforward line-by-line translation of these formulas into Scala, with some tests:

```
def integration(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)
}

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

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

The entire code is one large *expression*, with a few sub-expressions (`s1`, `s2`, etc.) defined for convenience in the **local scope** of the function (that is, within the function's body); the code contains no loops. This is similar to the way mathematical texts 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 then 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, where *every* sub-expression — that is, every step of every calculation — must be assigned 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 similarly useful if data structures can be created without names. For instance, a **dictionary** (also called a "map") 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 repetitive code that creates an initially empty dictionary and then fills it step by step with values:

```
// 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 nameless values of other types: they allow us to build larger programs from simpler parts in a uniform way.

1.7.6 Historical perspective on nameless functions

Nameless functions were first used in 1936 in a theoretical programming language called "λ-calculus". In that language,[9] all functions are nameless and have a single argument. The letter λ is a syntax separator denoting function arguments in nameless functions. For example, the nameless function $x \to x + 1$ could 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, most languages added support for nameless functions, because programming in the map/reduce style (which invites frequent use of nameless functions) turned out to be immensely productive. Table 1.2 shows the year when nameless functions were introduced in each language.

What this book calls a "nameless function" is also called anonymous function, function expression, function literal, closure, lambda function, lambda expression, or just a "lambda".

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))`
Standard ML	1973	`fn (k: int) => k + 1`
Caml	1985	`fun (k: int) -> k + 1`
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 programming languages.

[9] Although called a "calculus," it is a (drastically simplified) programming language, not related to differential or integral calculus. Also, the letter λ has no particular significance; it plays a purely syntactic role in the λ-calculus. Practitioners of functional programming usually do not need to study any λ-calculus. All practically relevant knowledge related to λ-calculus is explained in Chapter 4 of this book.

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 of 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 any number of values, which may be called **parts** of a tuple (they are also called **fields** of a tuple). The parts of a tuple can have different types, but the type of each part is fixed once and for all. Also, the number of parts in a tuple is fixed. It is a **type error** to use incorrect types in a tuple, or an incorrect number of parts of a tuple:

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

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

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

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

scala> a._1
res0: Int = 123

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

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

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

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

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

Tuples can be **nested**: any part of a tuple can be itself 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* `=> ...`

```
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
```

An example of a **destructuring definition** is shown at left. 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. The expression `{case (a, b, c) => ...}` called a **case expression** (shown at left) 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

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

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:

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

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

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

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

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

A shorter syntax for functions such as `{case (x, y) => y}` that extract elements from tuples `(t => t._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(t => t._2)
res2: Seq[Int] = List(10, 20, 30)
```

2.1.3 Using tuples with collections

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

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

We can create a list of tuples,

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

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

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

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

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

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

In this way, we can use the standard methods such as `map`, `filter`, `max`, `sum` to manipulate sequences of tuples. The names 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)
}
res2: List[(String, Boolean)] = List((apples,false), (pears,false), (lemons,true))
```

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

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

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

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

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

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

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

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

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

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

In the last line, the equivalent "dotless" infix syntax (`s zip t`) is shown 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 transforms a sequence into a sequence of pairs, where the second part of the pair is the zero-based index:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The `sliding` method creates a 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, -0.0011160413780774369, 0.003947079736951305,
    0.005381273944717851, 0.0018679497047270743)
```

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

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

scala> Seq("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 Solved examples: Tuples and collections

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

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

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

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

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

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

Solution The method `count` takes a predicate and returns the number of 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, compute the sequence of differences $c_i = a_i - b_i$.

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

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

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

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

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

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

Hint: use `zip` and `tail`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Hint: use `flatMap` and `toMap`.

Solution We are required to make a dictionary that maps pairs of integers `(x, y)` to `x * y`. Begin by creating the list of *keys* for that dictionary, which must be a list of pairs `(x, y)` of the form `List((1,1), (1,2), ..., (2,1), (2,2), ...)`. We need to 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 parentheses and write `(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))
```

It remains to convert this list of tuples to a dictionary with `toMap`. Also, for better readability, we can use Scala's pair syntax, `key -> value`, which is 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 some ‘Double‘ numbers.
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:

```
dict.map { case (name, addr) => (addr, name) }   // The result is converted to Map automatically.
```

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 procedure for reversing a dictionary works in the same way for dictionaries of any type. So the body of the function `rev` does not actually need to know the types of the keys and values in the dictionary. For this reason, it was easy for us to change the specific 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.

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

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

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

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

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

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

2.1.7 Exercises: Tuples and collections

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

Hint: use `flatMap` and `filter`.

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

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

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

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

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 required 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 type parameter 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 `String` and `Int`. Test it with `P = Boolean` and `Q = Set[Int]`.

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

```
Seq(("apple", 2), ("pear", 3), ("apple", 5), ("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[List[Int]]`, compute a new `Seq[List[Int]]` where each inner list contains *three* largest elements from the initial inner list (or fewer than three if the initial inner list is shorter).

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

Exercise 2.1.7.11 **(a)** Given two sets, `p: Set[Int]` and `q: Set[Int]`, compute a set of type `Set[(Int, Int)]` as the Cartesian product of the sets `p` and `q`; that is, the set of all pairs `(x, y)` where `x` is an element from the set `p` and `y` is an element from the set `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, Church paid 100 and Gentzen paid 50, etc.
scala> payments(Seq(Map("Tarski" -> 10, "Gödel" -> 20), Map("Church" -> 100, "Gentzen" -> 50),
     Map("Tarski" -> 50), 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`, `mapValues` on dictionaries.

2.2 Converting a sequence into a single value

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

A simple computation that is impossible to do using `map` is 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 solvable with `map`, `sum`, etc., is to compute a floating-point number from a given sequence of decimal digits (including a “dot” character):

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

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

Why is it impossible to implement this function using `map`, `sum`, and other methods we have seen so far? In fact, 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
```

This task is doable because 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, where 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. This is how induction works in mathematics, and it works in the same way in functional programming. With this approach, the inductive definition of the method `sum` looks like this:

- The sum of an empty sequence is 0. That is, `Seq().sum == 0`.

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

The inductive definition of the function `digitsToInt` is:

- For an empty sequence of digits, `Seq()`, the result is `0`. This is a convenient base case, even if we never call `digitsToInt` on an empty sequence.

- If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs :+ x` with one more digit `x`, then

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

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

- The length of a sequence:
  - for an empty sequence, `Seq().length == 0`
  - if `xs.length` is known then `(xs :+ x).length == xs.length + 1`

- Maximum element of a sequence (undefined for empty sequences):
  - for a one-element sequence, `Seq(x).max == x`
  - if `xs.max` is known then `(xs :+ x).max == math.max(xs.max, x)`

- Count the sequence elements satisfying a predicate `p`:
  - for an empty sequence, `Seq().count(p) == 0`
  - if `xs.count(p)` is known then `(xs :+ x).count(p) == xs.count(p) + c`, where we set `c = 1` when `p(x) == true` and `c = 0` otherwise

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

2.2.2 Implementing functions by recursion

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

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

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

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

We translate mathematical induction into code by first writing a condition to decide whether we 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. In this example, the `if/else` expression will separate 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]

```
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.
}
```

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.

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

```
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.

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 each inductively defined function repeats the code for checking the base case and the code for splitting the sequence `s` into the subsequence `xs` and the extra element `x`. This repeated common code can be put into a library function, and the Scala library provides such functions. We will look at using them in Section 2.2.4.

2.2.3 Tail recursion

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

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

scala> lengthS((1 to 1000).toList)
```

[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 (`+`) to a single element (`x`) that is being added.

```
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(...)`. So we can visualize how the computer evaluates this code:

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

The function body of `lengthS` will evaluate the inductive step, that is, the "`else`" part of the "`if/else`", about $100,000$ times. Each time, the sub-expression with nested computations `1+(1+(...))` will get larger. This intermediate sub-expression needs to be held somewhere in memory, until at some point the function body goes into the base case and returns a value. When that happens, the entire intermediate sub-expression will contain about $100,000$ nested function calls still waiting to be evaluated. This sub-expression is held in a special area of memory called **stack memory**, where the not-yet-evaluated nested function calls are held in the order of their calls on a stack. Due to the way computer memory is managed, the stack memory has a fixed size and cannot grow automatically. So, when the intermediate expression becomes large enough, it causes an overflow of the stack memory and 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.

```
@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`.

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

```
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):

```
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 far from 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. In code, it means that intermediate expressions are computed immediately before making recursive calls; this avoids the growth of the intermediate expressions.

Usually, the accumulator trick works because some associativity law is present. In that case, we are able to rearrange the order of recursive calls so that these calls always occur outside all other sub-expressions — that is, in tail positions. However, not all computations obey a suitable associativity law. Even if a code rearrangement exists, it may not be immediately obvious how to find it.

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

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

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

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

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

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

Despite a certain similarity between this code and the code of `digitsToInt` from the previous subsection, the implementation `fromDigits` cannot be directly derived from the inductive definition of `digitsToInt`. One needs a separate proof that `fromDigits(s, 0)` computes the same result as `digitsToInt(s)`. This holds due to 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 prove this by induction. 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 we need to prove that

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

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

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

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). The proof uses induction and begins by writing the code of d in a short notation,

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

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

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

The 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:

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

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 so 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 making `b0` and `g` into parameters. A possible implementation is

```
def f[A, B](s: Seq[A], b: B, g: (A, B) => B): B =
  if (s.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 is of 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.

The method `foldLeft` is available in the Scala library for all collections, including dictionaries and sets. Since `foldLeft` is tail-recursive, no stack overflows will occur even for very large sequences.

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

2.2.5 Solved 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 `math.max(b, x)`. So, the code is:

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

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

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

Example 2.2.5.2 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, Double)`. To avoid repeating a long type expression, we can define a type alias for it, say, `D4`:

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

The updater 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)
  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* the accumulator value. So, we need to update the accumulator by appending the value `f(x)`, where `x` is the current element of the input sequence:

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

To see what `digitsToDouble` must do, let us consider how the evaluation of `digitsToDouble(Seq('3', '4', '.', '2', '5'))` should go. We can write a table showing the intermediate result at each iteration. This will hopefully help us figure out what the accumulator and the updater function $g(...)$ must be:

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

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:

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

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 a dot character was seen.
    else {
      val digit = c - '0'
      if (flag) (num + digit / factor, flag, factor * 10) // This digit is after the dot.
      else (num * 10 + digit, flag, factor)               // This digit is before the dot.
    }
  }
```

Now we can implement `digitsToDouble` as follows:

```
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 the 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 required type signature and a sample test:

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

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

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

```
def byLength[A](xs: Seq[A], length: 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]]`. To work with dictionaries, you will need to use the methods `getOrElse` and `updated`. The method `getOrElse` fetches a value from a dictionary by key, and returns the given default value if the dictionary does not contain that key:

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

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

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

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

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

2.3 Converting a single value into a sequence

An aggregation converts ("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 of the stream until it runs out of memory and crashes.

Streams are similar to sequences, and methods such as `map`, `filter`, and `flatMap` are also defined for streams. 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 for, say, $n = 2405$, we need to divide n repeatedly by 10, getting a sequence n_k of intermediate numbers ($n_0 = 2405$, $n_1 = 240$, ...) and the corresponding sequence of last digits, $n_k \bmod 10$ (in this example: 5, 0, ...). The sequence n_k is defined using mathematical induction:

- Base case: $n_0 = n$, where n is 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 such as `(_ % 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
  }
```

[2] In a future version of Scala 3, the `Stream` class will be replaced by `LazyList`.

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: 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.

Definition by induction	Scala code example
$f([]) = b\,;\;\; f(s{+\!\!+}[x]) = g(f(s), x)$	`f(xs) = xs.foldLeft(b)(g)`
$x_0 = b\,;\;\; x_{k+1} = g(x_k)$	`xs = Stream.iterate(b)(g)`
$y_0 = b\,;\;\; y_{k+1} = g(y_k, x_k)$	`ys = xs.scanLeft(b)(g)`

Table 2.1: Implementing mathematical induction.

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 tasks are not possible with these tools? We cannot implement a non-tail-recursive function without stack overflow (i.e., without unlimited growth of intermediate expressions). The accumulator trick does not always work! In some cases, it is impossible to implement tail recursion in a given recursive computation. An example of such a computation is the "merge-sort" algorithm where the function body must contain two recursive calls within a single expression. (It is impossible to rewrite *two* recursive calls as one.)

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

2.5.1 Solved examples

Example 2.5.1.1 Compute the smallest n such that $f(f(f(...f(1)...) \geq 1000$, where the function f is applied n times. Write this as a function taking f, 1, and 1000 as arguments. Test with $f(x) = 2x + 1$.

Solution 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 plus 1:

```
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> 1 + Stream.iterate(1)(x => 2 * x + 1).takeWhile(x => x < 1000).length
res1: Int = 10
```

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 specialized 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 specialized data structures:

```
def maxKSoFar(xs: Stream[Int], k: Int): Stream[Seq[Int]] = {
  // The initial value of the accumulator is an empty Seq() of type Seq[Int].
  xs.scanLeft(Seq[Int]()) { (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).mapValues(_.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 `mapValues`:

```
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.mapValues(_.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[3] 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) Re-implement `binSearch` using `Stream.iterate` without writing explicitly recursive code.

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 $i + 1 = j$.

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

For convenience, let us introduce *two* accumulator values representing (i, 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, the code

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

will generate an error: the arguments in the same argument 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. The code becomes:

```
def binSearch(xs: Seq[Int], goal: Int): Int = {
  type Acc = (Int, Int)
  val init: Acc = (0, xs.length)
  val updater: Acc => Acc = { case (left, right) =>
    if (right - left <= 1 || 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 the element with tight enough bounds.
    .get._1                              // Take the ‘left‘ bound from that element.
}
```

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 = if (n == 0) 0 else 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 detecting repetition becomes quick:

```
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`. 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 must stop at `init`. (This is the base case of the induction). Otherwise, `next(init) == Some(x)` yields a new value `x` and indicates that we need to continue to "unfold" the stream with `x` instead of `init`. (This is the inductive step.) Streams can be created from individual values via the Scala standard library method `Stream.cons` that constructs a stream from a single value and a tail:

```
def unfold[A](init: A)(next: A => Option[A]): Stream[A] = next(init) match {
  case None      => Stream(init)                   // A stream containing a single value `init`.
  case Some(x)   => Stream.cons(init, unfold(x)(next)) // `init` followed by the tail of 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[4] 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())
```

[4] https://en.wikipedia.org/wiki/Dynamic_programming

```
  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 Compute the sum of squared digits of a given integer; e.g., `dsq(123) = 14` (see Example 2.5.1.6). Generalize the solution to take as an argument an 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[5] it will).

Exercise 2.5.2.3 For a given integer n, compute the sum of cubed digits, then the sum of cubed digits of the result, etc.; stop the resulting sequence when it repeats itself, and so determine whether it ever reaches 1. (Use Exercise 2.5.2.1.)

```
def cubes(n: Int): Stream[Int] = ???

scala> cubes(123).take(10).toList
res0: List[Int] = List(123, 36, 243, 99, 1458, 702, 351, 153, 153, 153)

scala> cubes(2).take(10).toList
res1: List[Int] = List(2, 8, 512, 134, 92, 737, 713, 371, 371, 371)

scala> cubes(4).take(10).toList
res2: List[Int] = List(4, 64, 280, 520, 133, 55, 250, 133, 55, 250)

def cubesReach1(n: Int): Boolean = ???

scala> cubesReach1(10)
res3: Boolean = true

scala> cubesReach1(4)
res4: Boolean = false
```

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

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 `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 (other integers k such that $k < n$ and n/k is an integer). For example, 6 is a perfect number because its divisors are 1, 2, and 3, and $1 + 2 + 3 = 6$, while 8 is not a perfect number because its divisors are 1, 2, and 4, and $1 + 2 + 4 = 7 \neq 8$. Write a function that determines whether a given number n is perfect. Determine all perfect numbers up to one million.

Exercise 2.5.2.12 Remove adjacent repeated elements from a sequence of type `Seq[A]` when they are repeated more than k times. Repetitions up to k times should remain unchanged. The required type signature and a sample test:

```
def removeDups[A](s: Seq[A], k: Int): Seq[A] = ???

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

Exercise 2.5.2.13 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 re-implements 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 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.

A simple example of a partial function in Scala is the `max` method: it only works for non-empty sequences. Trying to evaluate it on an empty sequence generates an error called an "exception":

```
scala> Seq(1).tail
res0: Seq[Int] = List()
scala> res0.max
java.lang.UnsupportedOperationException: empty.max
  at scala.collection.TraversableOnce$class.max(TraversableOnce.scala:229)
  at scala.collection.AbstractTraversable.max(Traversable.scala:104)
  ... 32 elided
```

This kind of error may crash the entire program at run time. Unlike the type errors we saw before, which occur at compilation time (i.e., before the program can start), **run-time errors** occur while the program is running, and only when some partial function happens to get an incorrect input. The incorrect input may occur at any point after the program started running, which may crash the entire program in the middle of a long computation.

So, it seems clear that we should write code that does not generate such errors. For instance, it is safe to apply `max` to a sequence if we know that it is non-empty.

Sometimes, a function that uses pattern matching turns out to be a partial function because its pattern matching code fails on certain input data.

If a pattern matching expression fails, the code will throw an exception and stop running. In functional programming, we usually want to avoid this situation because it makes it much harder to reason about program correctness. In most cases, programs can be written to avoid the possibility of match errors. An example of an unsafe pattern matching expression is

```
def h(p: (Int, Int)): Int = p match { case (x, 0) => x }

scala> h( (1,0) )
res0: Int = 1

scala> h( (1,2) )
 scala.MatchError: (1,2) (of class scala.Tuple2$mcII$sp)
  at .h(<console>:12)
  ... 32 elided
```

Here the pattern contains a pattern variable `x` and a constant `0`. This pattern only matches tuples whose second part is equal to `0`. If the second argument is nonzero, a match error occurs and the program crashes. So, `h` is a partial function.

Pattern matching failures never happen if we match a tuple of correct size with a pattern such as `(x, y, z)`, because a pattern variable will always match a value. So, pattern matching with a pattern such as `(x, y, z)` is **infallible** (never fails at run time) when applied to a tuple with 3 elements.

Another way in which pattern matching can be made infallible is by including a pattern that matches everything:

```
p match {
  case (x, 0)   => ...   // This only matches some tuples.
  case _        => ...   // This matches everything.
}
```

If the first pattern `(x, 0)` fails to match the value `p`, the second pattern will be tried (and will always succeed). The `case` patterns in a `match` expression are tried in the order they are written. So, a `match` expression may be made infallible by adding a "match-all" underscore pattern.

2.6.2 Scope and shadowing of pattern matching variables

Pattern matching introduces **locally scoped** variables — that is, variables defined only on the right-hand side of the pattern match expression. As an example, consider this code:

```
def f(x: (Int, Int)): Int = x match { case (x, y) => x + y }

scala> f( (2,4) )
res0: Int = 6
```

The argument of `f` is the variable `x` of a tuple type `(Int,Int)`, but there is also a pattern variable `x` in the case expression. The pattern variable `x` matches the first part of the tuple and has type `Int`. Because variables are locally scoped, the pattern variable `x` is only defined within the expression `x + y`. The argument `x:(Int,Int)` is a completely different variable whose value has a different type.

The code works correctly but is confusing to read because of the name clash between the two quite different variables, both named `x`. Another negative consequence of the name clash is that the argument `x:(Int,Int)` *is invisible* within the case expression: if we write "`x`" in that expression, we will get the pattern variable `x:Int`. One says that the argument `x:(Int,Int)` has been **shadowed** by the pattern variable `x` (which is a "bound variable" inside the case expression).

The problem is easy to avoid: we can give the pattern variable another name. Since the pattern variable is locally scoped, it can be renamed within its scope without affecting any other code:

```
def f(x: (Int, Int)): Int = x match { case (a, b) => a + b }

scala> f( (2,4) )
res0: Int = 6
```

2.6.3 Lazy values and sequences. Iterators and streams

We have used streams to create sequences whose length is not known in advance. An example is a stream containing a sequence of increasing positive integers:

```
scala> val p = Stream.iterate(1)(_ + 1)
p: Stream[Int] = Stream(1, ?)
```

At this point, we have not defined a stopping condition for this stream. In some sense, streams may be seen as "infinite" sequences, although in practice a stream is always finite because computers cannot run infinitely long. Also, computers cannot store infinitely many values in memory.

More precisely, streams are "partially computed" rather than "infinite". The main difference between arrays and streams is that a stream's elements are computed on demand and not all initially available, while an array's elements are all computed in advance and are immediately available.

Generally, there are four possible ways a value could be available:

Availability	Explanation	Example Scala code
"eager"	computed immediately	`val z = f(123)`
"lazy"	computed upon first use	`lazy val z = f(123)`
"on-call"	computed each time it is used	`def z = f(123)`
"never"	cannot be computed due to errors	`val (x, y) = "abc"`

A **lazy value** (declared as `lazy val` in Scala) is computed only when it is needed in some other expression. Once computed, a lazy value stays in memory and will not be re-computed.

An "on-call" value is re-computed every time it is used. In Scala, a `def` declaration does that.

Most collection types in Scala (such as `List`, `Array`, `Set`, and `Map`) are **eager**: all elements of an eager collection are already evaluated.

A stream is a **lazy collection**. Elements of a stream are computed when first needed; after that, they remain in memory and will not be computed again:

```
scala> val str = Stream.iterate(1)(_ + 1)
str: Stream[Int] = Stream(1, ?)

scala> str.take(10).toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> str
res1: Stream[Int] = Stream(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ?)
```

In many cases, it is not necessary to keep previous values of a sequence in memory. For example:

```
scala> (1L to 1000000000L).sum        // Compute the sum of integers from 1 to 1 billion.
res0: Long = 500000000500000000
```

We do not actually need to put a billion numbers in memory if we only want to compute their sum. Indeed, the computation just shown does *not* put all the numbers in memory. The computation will fail if we use a list or a stream:

```
scala> (1L to 1000000000L).toStream.sum
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

The code `(1L to 1000000000L).sum` works because `(1 to n)` produces a sequence whose elements are computed whenever needed but do not remain in memory. This can be seen as a sequence with the "on-call" availability of elements. Sequences of this sort are called **iterators**:

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 1 until 5
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

The types `Range` and `Range.Inclusive` are defined in the Scala standard library and are iterators. They behave as collections and support the usual methods (`map`, `filter`, etc.), but they do not store previously computed values in memory.

The `view` method Eager collections such as `List` or `Array` can be converted to iterators by using the `view` method. This is necessary when intermediate collections consume too much memory when fully evaluated. For example, consider the computation of Example 2.1.5.7 where we used `flatMap` to replace each element of an initial sequence by three new numbers before computing `max` of the resulting collection. If instead of three new numbers we wanted to compute *three million* new numbers each time, the intermediate collection created by `flatMap` would require too much memory, and the computation would crash:

```
scala> (1 to 10).flatMap(x => 1 to 3000000).max
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Even though the range `(1 to 10)` is an iterator, a subsequent `flatMap` operation creates an intermediate collection that is too large for our computer's memory. We can use `view` to avoid this:

```
scala> (1 to 10).view.flatMap(x => 1 to 3000000).max
res0: Int = 3000000
```

The choice between using streams and using iterators is dictated by memory constraints. Except for that, streams and iterators behave similarly to other sequences. We may write programs in the map/reduce style, applying standard methods such as `map`, `filter`, etc., to streams and iterators. Mathematical reasoning about transforming a sequence is the same, whether the sequence is eager, lazy, or on-call.

The `Iterator` class The Scala library class `Iterator` has methods such as `Iterator.iterate` and others, similarly to `Stream`. However, `Iterator` does not behave as a *value* in the mathematical sense:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> iter.toList // Look at the elements of ‘iter‘.
```

```
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> iter.toList // Look at those elements again...??
res1: List[Int] = List()

scala> iter
res2: Iterator[Int] = empty iterator
```

Evaluating the expression `iter.toList` two times produces a different result the second time. As we see from the Scala output, the value `iter` has become "empty" after the first use.

This situation is impossible in mathematics: if x is some value, such as 100, and f is some function, such as $f(x) = \sqrt{x}$, then $f(x)$ will be the same, $f(100) = \sqrt{100} = 10$, no matter how many times we compute $f(x)$. For instance, we can compute $f(x) + f(x) = 20$ and obtain the correct result. We could also set $y = f(x)$ and compute $y + y = 20$, with the same result. This property is called **referential transparency** or **functional purity** of the function f. After applying a pure function, we can be sure that, for instance, no hidden values in memory have been modified.

When we set $x = 100$ and compute $f(x) + f(x)$, the number 100 does not "become empty" after the first use; its value remains the same. This behavior is called the **value semantics** of numbers. One says that integers "are values" in the mathematical sense. Alternatively, one says that numbers are **immutable,** i.e., cannot be changed. (What would it mean to "modify" the number 10?)

In programming, a type has value semantics if a given computation applied to it always gives the same result. Usually, this means that the type contains immutable data, and the computation is referentially transparent. We can see that Scala's `Range` has value semantics and is immutable:

```
scala> val x = 1 until 10
x: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> x.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Collections such as `List`, `Map`, or `Stream` are immutable. Some elements of a `Stream` may not be evaluated yet, but this does not affect its value semantics:

```
scala> val str = (1 until 10).toStream
str: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Iterators produced by applying the `view` method to collections will have value semantics:

```
scala> val v = (1 until 10).view
v: scala.collection.SeqView[Int,IndexedSeq[Int]] = SeqView(...)

scala> v.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> v.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Due to the lack of value semantics, programs written using `Iterator` may not obey the usual rules of mathematical reasoning. This makes it easy to write wrong code that looks correct.

To illustrate the problem, let us re-implement Example 2.5.1.9 by keeping the same code but using `Iterator` instead of `Stream`:

```
def stopRepeatsBad[T](iter: Iterator[T]): Iterator[T] = {
  val halfSpeed = iter.flatMap(x => Seq(x, x))
```

```
  halfSpeed.zip(iter) // Do not prepend the first element. It won't help.
  .drop(1).takeWhile { case (h, s) => h != s }
  .map(_._2)
}

scala> stopRepeatsBad(Seq(1, 3, 5, 7, 9, 3, 5, 7, 9).toIterator).toList
res0: List[Int] = List(5, 9, 3, 7, 9)
```

The result $[5,9,3,7,9]$ is incorrect, but not in an obvious way: the sequence *was* stopped at a repetition, as we wanted, but some of the elements of the given sequence are missing (while other elements are present). It is difficult to debug a program that produces *partially* correct numbers.

The error in this code occurs in the expression `halfSpeed.zip(iter)` due to the fact that `halfSpeed` was itself defined via `iter`. The result is that `iter` is used twice in this code, which leads to errors because `iter` is mutable and does not behave as a value. Creating an `Iterator` and using it twice in the same expression can give wrong results or even fail with an exception:

```
scala> val s = (1 until 10).toIterator
s: Iterator[Int] = non-empty iterator

scala> val t = s.zip(s).toList
java.util.NoSuchElementException: next on empty iterator
```

It is surprising and counter-intuitive that a variable cannot be used twice in some expression. Intuitively, we expect code such as `s.zip(s)` to work correctly even though the variable `s` is used twice. When we read the expression `s.zip(s)`, we imagine a given sequence `s` being "zipped" with itself. So we reason that `s.zip(s)` should produce a sequence of pairs. But Scala's `Iterator` is **mutable** (can be modified during use), which breaks the usual ways of mathematical reasoning about code.

The self-modifying behavior of `Iterator` is an example of a side effect. A function has a **side effect** if the function's code performs some action in addition to computing the result value. Examples of side effects are: starting and stopping external processes; modifying values stored in memory; writing files; printing; sending or receiving data over a network; and playing sounds. Functions with side effects do not have value semantics. Calling such a function twice produces the side effect twice, which is not the same as calling the function once and simply re-using the result value. On the other hand, pure functions have no side effects and have value semantics.

An `Iterator` can be converted to a `Stream` using the `toStream` method. This restores the value semantics, since streams are values:

```
scala> val iter = (1 until 10).toIterator
iter: Iterator[Int] = non-empty iterator

scala> val str = iter.toStream
str: Stream[Int] = Stream(1, ?)

scala> str.toList
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.toList
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> str.zip(str).toList
res2: List[(Int, Int)] = List((1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9))
```

Instead of `Iterator`, we can use `Stream` and `view` when lazy or on-call collections are required.

Libraries such as `scalaz` and `fs2` also provide lazy and on-call streams with correct value semantics.

3 The logic of types. I. Disjunctive types

Disjunctive types describe values that belong to a disjoint set of alternatives.

To see how Scala implements disjunctive types, we need to begin by looking at "case classes".

3.1 Scala's case classes

3.1.1 Tuple types with names

It is often helpful to use names for the different parts of a tuple. Suppose that some program represents the size and the color of socks with the tuple type `(Double, String)`. What if the same tuple type `(Double, String)` is used in another place in the program to mean the amount paid and the payee? A programmer could mix the two values by mistake, and it would be hard to find out why the program incorrectly computes, say, the total amount paid.

```
def totalAmountPaid(ps: Seq[(Double, String)]): Double = ps.map(_._1).sum
val x = (10.5, "white")       // Sock size and color.
val y = (25.0, "restaurant")  // Payment amount and payee.

scala> totalAmountPaid(Seq(x, y)) // Nonsense.
res0: Double = 35.5
```

We would prevent this kind of mistake if we could use two *different* types, with names such as `MySock` and `Payment`, for the two kinds of data. There are three basic ways of defining a new named type in Scala: using a type alias, using a class (or "trait"), and using an opaque type.

Opaque types (hiding a type under a new name) is a feature of a future version of Scala 3; so we focus on type aliases and case classes.

A **type alias** is an alternative name for an existing (already defined) type. We could use type aliases in our example to add clarity to the code:

```
type MySockTuple = (Double, String)
type PaymentTuple = (Double, String)

scala> val s: MySockTuple = (10.5, "white")
s: MySockTuple = (10.5,white)

scala> val p: PaymentTuple = (25.0, "restaurant")
p: PaymentTuple = (25.0,restaurant)
```

But type aliases do not prevent mix-up errors:

```
scala> totalAmountPaid(Seq(s, p)) // Nonsense again.
res1: Double = 35.5
```

Scala's **case classes** can be seen as "tuples with names". A case class is equivalent to a tuple type that has a name chosen when we define the case class. Also, each part of the case class will have a separate name that we must choose. This is how to define case classes for the example with socks and payments:

```
case class MySock(size: Double, color: String)
case class Payment(amount: Double, name: String)

scala> val sock = MySock(10.5, "white")
sock: MySock = MySock(10.5,white)
```

```
scala> val paid = Payment(25.0, "restaurant")
paid: Payment = Payment(25.0,restaurant)
```

This code defines new types named `MySock` and `Payment`. Values of type `MySock` are written as `MySock(10.5, "white")`, which is similar to writing the tuple `(10.5, "white")` except for adding the name `MySock` in front of the tuple.

To access the parts of a case class, we use the part names:

```
scala> sock.size
res2: Double = 10.5

scala> paid.amount
res3: Double = 25.0
```

The mix-up error is now a type error detected by the compiler:

```
def totalAmountPaid(ps: Seq[Payment]): Double = ps.map(_.amount).sum

scala> totalAmountPaid(Seq(paid, paid))
res4: Double = 50.0

scala> totalAmountPaid(Seq(sock, paid))
<console>:19: error: type mismatch;
 found   : MySock
 required: Payment
       totalAmountPaid(Seq(sock, paid))
                           ^
```

A function whose argument is of type `MySock` cannot be applied to an argument of type `Payment`. Case classes with different names are *different types*, even if they contain the same parts.

Just as tuples can have any number of parts, case classes can have any number of parts, but the part names must be distinct, for example:

```
case class Person(firstName: String, lastName: String, age: Int)

scala> val noether = Person("Emmy", "Noether", 137)
noether: Person = Person(Emmy,Noether,137)

scala> noether.firstName
res5: String = Emmy

scala> noether.age
res6: Int = 137
```

This data type carries the same information as a tuple `(String, String, Int)`. However, the declaration of a `case class Person` gives the programmer several features that make working with the tuple's data more convenient and less error-prone.

Some (or all) part names may be specified when creating a case class value:

```
scala> val poincaré = Person(firstName = "Henri", lastName = "Poincaré", 165)
poincaré: Person = Person(Henri,Poincaré,165)
```

It is a type error to use wrong types with a case class:

```
scala> val p = Person(140, "Einstein", "Albert")
<console>:13: error: type mismatch;
 found   : Int(140)
 required: String
       val p = Person(140, "Einstein", "Albert")
                      ^
<console>:13: error: type mismatch;
 found   : String("Albert")
 required: Int
       val p = Person(140, "Einstein", "Albert")
                                       ^
```

This error is due to an incorrect order of parts when creating a case class value. However, parts can be specified in any order when using part names:

```
scala> val p = Person(age = 137, lastName = "Noether", firstName = "Emmy")
p: Person = Person(Emmy,Noether,137)
```

A part of a case class can have the type of another case class, creating a type similar to a nested tuple:

```
case class BagOfSocks(sock: MySock, count: Int)
val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> bag.sock.size
res7: Double = 10.5
```

3.1.2 Case classes with type parameters

Type classes can be defined with type parameters. As an example, consider an extension of `MySock` where, in addition to the size and color, an "extended sock" holds another value. We could define several specialized case classes,

```
case class MySock_Int(size: Double, color: String, value: Int)
case class MySock_Boolean(size: Double, color: String, value: Boolean)
```

but it is better to define a single parameterized case class

```
case class MySockX[A](size: Double, color: String, value: A)
```

This case class can accommodate every type `A`. We may now create values of `MySockX` containing a `value` of any given type, say `Int`:

```
scala> val s = MySockX(10.5, "white", 123)
s: MySockX[Int] = MySockX(10.5,white,123)
```

Because the value `123` has type `Int`, the type parameter `A` in `MySockX[A]` was automatically set to the type `Int`. The result has type `MySockX[Int]`. The programmer does not need to specify that type explicitly.

Each time we create a value of type `MySockX`, a specific type will have to be used instead of the type parameter `A`. If we want to be explicit, we may write the type parameter like this:

```
scala> val s = MySockX[String](10.5, "white", "last pair")
s: MySockX[String] = MySockX(10.5,white,last pair)
```

However, we can write **parametric code** working with `MySockX[A]`, that is, keeping the type parameter `A` in the code. For example, a function that checks whether a sock of type `MySockX[A]` fits the author's foot can be written as

```
def fits[A](sock: MySockX[A]): Boolean = (sock.size >= 10.5 && sock.size <= 11.0)
```

This function is defined for all types `A` at once, because its code works in the same way regardless of what `A` is. Scala will set the type parameter `A` automatically when we apply `fits` to an argument:

```
scala> fits(MySockX(10.5, "blue", List(1,2,3))) // Type parameter A = List[Int].
res0: Boolean = true
```

This code forces the type parameter `A` to be `List[Int]`, and so we may omit the type parameter of `fits`. When types become more complicated, it may be helpful to write out some type parameters. The compiler can detect a mismatch between the type parameter `A = List[Int]` used in the "sock" value and the type parameter `A = Int` in the function `fits`:

```
scala> fits[Int](MySockX(10.5, "blue", List(1,2,3)))
<console>:15: error: type mismatch;
 found   : List[Int]
 required: Int
       fits[Int](MySockX(10.5, "blue", List(1,2,3)))
                                           ^
```

Case classes may have several type parameters, and the types of the parts may use these type parameters. Here is an artificial example of a case class using type parameters in different ways,

```
case class Complicated[A,B,C,D](x: (A, A), y: (B, Int) => A, z: C => C)
```

This case class contains parts of different types that use the type parameters `A`, `B`, `C` in tuples and functions. The type parameter `D` is not used at all; this is allowed (and occasionally useful).

A type with type parameters, such as `MySockX` or `Complicated`, is called a **type constructor**. A type constructor "constructs" a new type, such as `MySockX[Int]`, from a given type parameter `Int`. Values of type `MySockX` cannot be created without setting the type parameter. So, it is important to distinguish the type constructor, such as `MySockX`, from a type that can have values, such as `MySockX[Int]`.

3.1.3 Tuples with one part and with zero parts

Let us compare tuples and case classes more systematically.

Parts of a case class are accessed by name with a dot syntax, for example `sock.color`. Parts of a tuple are accessed with the accessors such as `x._1`. This syntax is the same as that for a case class whose parts have names `_1`, `_2`, etc. So, it appears that tuple parts *do* have names in Scala, although those names are always automatically chosen as `_1`, `_2`, etc. Tuple types are also automatically named in Scala as `Tuple2`, `Tuple3`, etc., and they are parameterized, since each part of the tuple may be of any chosen type. A tuple type expression such as `(Int, String)` is just a special syntax for the parameterized type `Tuple2[Int, String]`. One could define the tuple types as case classes like this,

```
case class Tuple2[A, B](_1: A, _2: B)
case class Tuple3[A, B, C](_1: A, _2: B, _3: C)  // And so on with Tuple4, Tuple5...
```

if these types were not already defined in the Scala library.

Proceeding systematically, we ask whether tuple types can have just one part or even no parts. Indeed, Scala defines `Tuple1[A]` (which is rarely used in practice) as a tuple with a single part.

The tuple with zero parts also exists and is called `Unit` (rather than "`Tuple0`"). The syntax for the value of the `Unit` type is the empty tuple, `()`. It is clear that there is *only one* value, `()`, of this type; this explains the name "unit".

At first sight, the `Unit` type — a tuple that contains no data — may appear to be useless. It turns out, however, that the `Unit` type is important in functional programming. It is used as a type *guaranteed* to have only a single distinct value. This chapter will show some examples of using `Unit`.

Case classes may have one part or zero parts, similarly to the one-part and zero-part tuples:

```
case class B(z: Int)     // Tuple with one part.
case class C()           // Tuple with no parts.
```

The following table summarizes the correspondence between tuples and case classes:

Tuples	**Case classes**
`(123, "xyz"): Tuple2[Int, String]`	`case class A(x: Int, y: String)`
`(123,): Tuple1[Int]`	`case class B(z: Int)`
`(): Unit`	`case class C()`

Scala has an alternative syntax for empty case classes:

```
case object C   // Similar to ‘case class C()‘.
```

There are two main differences between `case class C()` and `case object C`:

- A `case object` cannot have type parameters, while we may define a `case class C[X, Y, Z]()` with type parameters `X`, `Y`, `Z` if needed.
- A `case object` is allocated in memory only once, while new values of a `case class C()` will be allocated in memory each time `C()` is evaluated.

Other than that, `case class C()` and `case object C` have the same meaning: a named tuple with zero parts, which we may also view as a "named `Unit`" type. This book will not use `case object`s because `case class`es are sufficient.

3.1.4 Pattern matching for case classes

Scala performs pattern matching in two situations:

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

Case classes can be used in both situations. A destructuring definition can be used in a function whose argument is of case class type `BagOfSocks`:

```
case class MySock(size: Double, color: String)
case class BagOfSocks(sock: MySock, count: Int)

def printBag(bag: BagOfSocks): String = {
  val BagOfSocks(MySock(size, color), count) = bag // Destructure the ‘bag‘.
  s"bag has $count $color socks of size $size"
}

val bag = BagOfSocks(MySock(10.5, "white"), 6)

scala> printBag(bag)
res0: String = bag has 6 white socks of size 10.5
```

A `case` expression can match a value, extract some pattern variables, and compute a result:

```
def fits(bag: BagOfSocks): Boolean = bag match {
  case BagOfSocks(MySock(size, _), _) => (size >= 10.5 && size <= 11.0)
}
```

In the code of this function, we match the `bag` value against the pattern `BagOfSocks(MySock(size, _), _)`. This pattern will always match and will define `size` as a pattern variable of type `Double`.

The syntax for pattern matching for case classes is similar to the syntax for pattern matching for tuples, except for the presence of the *names* of the case classes. For example, by removing the case class names from the pattern

```
case BagOfSocks(MySock(size, _), _) => ...
```

we obtain the nested tuple pattern

```
case ((size, _), _) => ...
```

that could be used for values of type `((Double, String), Int)`. We see that within pattern matching expressions, case classes behave as tuple types with added names.

Scala's "case classes" got their name from their use in `case` expressions. It is usually more convenient to use `case` expressions with case classes than to use destructuring definitions.

3.2 Disjunctive types

3.2.1 Motivation and first examples

In many situations, it is useful to have several different shapes of data within the same type. As a first example, suppose we are looking for real roots of a quadratic equation $x^2 + bx + c = 0$. There are three cases: no real roots, one real root, and two real roots. It is convenient to have a type that represents "real roots of a quadratic equation"; call it `RootsOfQ`. Inside that type, we distinguish between the three cases, but outside it looks like a single type.

Another example is the binary search algorithm that looks for an integer x in a sorted array. Either the algorithm finds the location of x in the array, or it determines that the array does not contain x. It is convenient if the algorithm could return a value of a single type (say, `SearchResult`) that represents *either* an index at which x is found, *or* the absence of an index.

More generally, we may have computations that *either* return a result *or* generate an error and fail to produce a result. It is then convenient to return a value of a single type (say, `Result`) that represents either a correct result or an error message.

In certain computer games, one has different types of "rooms", each room having certain properties depending on its type. Some rooms are dangerous because of monsters, other rooms contain useful objects, certain rooms allow you to finish the game, and so on. We want to represent all the different kinds of rooms uniformly, as a type `Room`, so that a value of type `Room` automatically stores the correct properties in each case.

In all these situations, data comes in several mutually exclusive shapes. This data can be represented by a single type if that type is able to describe a mutually exclusive set of cases:

- `RootsOfQ` must be either the empty tuple `()`, or `Double`, or a tuple `(Double, Double)`
- `SearchResult` must be either `Int` or the empty tuple `()`
- `Result` must be either an `Int` value or a `String` message

We see that the empty tuple, also known as the `Unit` type, is natural to use in these situations. It is also helpful to assign names to each of the cases:

- `RootsOfQ` is "no roots" with value `()`, or "one root" with value `Double`, or "two roots" with value `(Double, Double)`
- `SearchResult` is "index" with value `Int`, or "not found" with value `()`
- `Result` is "value" of type `Int` or "error message" of type `String`

Scala's case classes provide exactly what we need here — named tuples with zero, one, two or more parts. So it is natural to use case classes instead of tuples:

- `RootsOfQ` is a value of type `case class NoRoots()`, or a value of type `case class OneRoot(x: Double)`, or a value of type `case class TwoRoots(x: Double, y: Double)`
- `SearchResult` is a value of type `case class Index(Int)` or a value of type `case class NotFound()`
- `Result` is a value of type `case class Value(x: Int)` or a value of type `case class Error(message: String)`

Our three examples are now described as types that select one case class out of a given set. It remains to see how Scala defines such types. For instance, the definition of `RootsOfQ` needs to indicate that the case classes `NoRoots`, `OneRoot`, and `TwoRoots` are exactly the three alternatives described by the type `RootsOfQ`. The Scala syntax for that definition looks like this:

```
sealed trait RootsOfQ
final case class NoRoots()                       extends RootsOfQ
final case class OneRoot(x: Double)              extends RootsOfQ
final case class TwoRoots(x: Double, y: Double)  extends RootsOfQ
```

In the definition of `SearchResult`, we have two cases:

```
sealed trait SearchResult
final case class Index(i: Int)   extends SearchResult
final case class NotFound()      extends SearchResult
```

The definition of the `Result` type is parameterized, so that we can describe results of any type (while error messages are always of type `String`):

```
sealed trait Result[A]
final case class Value[A](x: A)              extends Result[A]
final case class Error[A](message: String)   extends Result[A]
```

The "`sealed trait` / `final case class`" syntax defines a type that represents a choice of one case class from a fixed set of case classes. This kind of type is called a **disjunctive type** in this book.

3.2.2 Solved examples: Pattern matching for disjunctive types

Our first examples of disjunctive types are `RootsOfQ`, `SearchResult`, and `Result[A]` defined in the previous section. We will now look at the Scala syntax for creating values of disjunctive types and for using the created values.

Consider the disjunctive type `RootsOfQ` having three case classes (`NoRoots`, `OneRoot`, `TwoRoots`). The only way of creating a value of type `RootsOfQ` is to create a value of one of these case classes. This is done by writing expressions such as `NoRoots()`, `OneRoot(2.0)`, or `TwoRoots(1.0, -1.0)`. Scala will accept these expressions as having the type `RootsOfQ`:

```
scala> val x: RootsOfQ = OneRoot(2.0)
x: RootsOfQ = OneRoot(2.0)
```

Given a value `x:RootsOfQ`, how can we use it, say, as a function argument? The main tool for working with values of disjunctive types is pattern matching. In Chapter 2, we used pattern matching with syntax such as `{ case (x, y) => ... }`. To use pattern matching with disjunctive types, we write *several* `case` patterns because we need to match several possible cases of the disjunctive type:

```
def f(r: RootsOfQ): String = r match {
  case NoRoots()       => "no real roots"
  case OneRoot(r)      => s"one real root: $r"
  case TwoRoots(x, y)  => s"real roots: ($x, $y)"
}

scala> f(x)
res0: String = "one real root: 2.0"
```

If we only need to recognize certain cases of a disjunctive type, we can match all other cases with an underscore:

```
scala> x match {
  case OneRoot(r)   => s"one real root: $r"
  case _            => "have something else"
}
res1: String = one real root: 2.0
```

The `match`/`case` expression represents a choice over possible values of a given type. Note the similarity with this code:

```
def f(x: Int): Int = x match {
  case 0    => println(s"error: must be nonzero"); -1
  case 1    => println(s"error: must be greater than 1"); -1
  case _    => x
}
```

The values `0` and `1` are some possible values of type `Int`, just as `OneRoot(1.0)` is a possible value of type `RootsOfQ`. When used with disjunctive types, `match`/`case` expressions will usually contain a complete list of possibilities. If the list of cases is incomplete, the Scala compiler will print a warning:

```
scala> def g(x: RootsOfQ): String = x match {
         case OneRoot(r) => s"one real root: $r"
      }
<console>:14: warning: match may not be exhaustive.
It would fail on the following inputs: NoRoots(), TwoRoots(_, _)
```

This code defines a *partial* function `g` that can be applied only to values of the form `OneRoot(...)` and will fail for other values.

Let us look at more examples of using the disjunctive types we just defined.

Example 3.2.2.1 Given a sequence of quadratic equations, compute a sequence containing their real roots as values of type `RootsOfQ`.

Solution Define a case class representing a quadratic equation $x^2 + bx + c = 0$:

```
case class QEqu(b: Double, c: Double)
```

The following function determines how many real roots an equation has:

```
def solve(quadraticEqu: QEqu): RootsOfQ = {
   val QEqu(b, c) = quadraticEqu // Destructure QEqu.
   val d = b * b / 4 - c
   if (d > 0) {
     val s = math.sqrt(d)
     TwoRoots(- b / 2 - s, - b / 2 + s)
   } else if (d == 0.0) OneRoot(- b / 2)
   else NoRoots()
}
```

Test the `solve` function:

```
scala> solve(QEqu(1,1))
res1: RootsOfQ = NoRoots()

scala> solve(QEqu(1,-1))
res2: RootsOfQ = TwoRoots(-1.618033988749895,0.6180339887498949)

scala> solve(QEqu(6,9))
res3: RootsOfQ = OneRoot(-3.0)
```

We can now implement the function `findRoots`,

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map(solve)
```

If the function `solve` will not be used often, we may want to write it inline as a nameless function:

```
def findRoots(equs: Seq[QEqu]): Seq[RootsOfQ] = equs.map { case QEqu(b, c) =>
  (b * b / 4 - c) match {
    case d if d > 0   =>
      val s = math.sqrt(d)
      TwoRoots(- b / 2 - s, - b / 2 + s)
    case 0.0          => OneRoot(- b / 2)
    case _            => NoRoots()
  }
}
```

This code depends on some features of Scala syntax. We can use the partial function `{ case QEqu(b, c) => ... }` directly as the argument of `map` instead of defining this function separately. This avoids having to destructure `QEqu` at a separate step. The `if`/`else` expression is replaced by an "embedded" `if` within the `case` expression, which is easier to read. Test the final code:

```
scala> findRoots(Seq(QEqu(1,1), QEqu(2,1)))
res4: Seq[RootsOfQ] = List(NoRoots(), OneRoot(-1.0))
```

Example 3.2.2.2 Given a sequence of values of type `RootsOfQ`, compute a sequence containing only the single roots. Example test:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = ???

scala> singleRoots(Seq(TwoRoots(-1, 1), OneRoot(3.0), OneRoot(1.0), NoRoots()))
res5: Seq[Double] = List(3.0, 1.0)
```

Solution We apply `filter` and `map` to the sequence of roots:

```
def singleRoots(rs: Seq[RootsOfQ]): Seq[Double] = rs.filter {
  case OneRoot(x) => true
  case _          => false
}.map { case OneRoot(x) => x }
```

In the `map` operation, we need to cover only the one-root case because the two other possibilities have been excluded ("filtered out") by the preceding `filter` operation.

Example 3.2.2.3 Implement binary search returning a `SearchResult`. Modify the binary search implementation from Example 2.5.1.5(b) so that it returns a `NotFound` value when appropriate.

Solution The code from Example 2.5.1.5(b) will return *some* index even if the given number is not present in the array:

```
scala> binSearch(Array(1, 3, 5, 7), goal = 5)
res6: Int = 2

scala> binSearch(Array(1, 3, 5, 7), goal = 4)
res7: Int = 1
```

In that case, the array's element at the computed index will not be equal to `goal`. We should return `NotFound()` in that case. The new code can be written as a `match`/`case` expression for clarity:

```
def safeBinSearch(xs: Seq[Int], goal: Int): SearchResult =
  binSearch(xs, goal) match {
    case n if xs(n) == goal   => Index(n)
    case _                    => NotFound()
  }
```

To test:

```
scala> safeBinSearch(Array(1, 3, 5, 7), 5)
res8: SearchResult = Index(2)

scala> safeBinSearch(Array(1, 3, 5, 7), 4)
res9: SearchResult = NotFound()
```

Example 3.2.2.4 Use the disjunctive type `Result[Int]` to implement "safe integer arithmetic", where a division by zero or a square root of a negative number will give an error message. Define arithmetic operations directly for values of type `Result[Int]`. When errors occur, abandon further computations.

Solution Begin by implementing the square root:

```
def sqrt(r: Result[Int]): Result[Int] = r match {
  case Value(x) if x >= 0  => Value(math.sqrt(x).toInt)
  case Value(x)            => Error(s"error: sqrt($x)")
  case Error(m)            => Error(m) // Keep the error message.
}
```

The square root is computed only if we have the `Value(x)` case, and only if $x \geq 0$. If the argument `r` was already an `Error` case, we keep the error message and perform no further computations.

To implement the addition operation, we need a bit more work:

```
def add(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x + y)
  case (Error(m), _)        => Error(m)  // Keep the first error message.
  case (_, Error(m))        => Error(m)  // Keep the second error message.
}
```

This code illustrates nested patterns that match the tuple `(rx, ry)` against various possibilities. In this way, the code is clearer than code written with nested `if`/`else` expressions.

Implementing the multiplication operation results in almost the same code:

```
def mul(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) => Value(x * y)
  case (Error(m), _)        => Error(m)
  case (_, Error(m))        => Error(m)
```

```
}
```

To avoid repetition, we may define a general function that "maps" operations on integers to operations on `Result[Int]` types:

```
def map2(rx: Result[Int], ry: Result[Int])(op: (Int, Int) => Int): Result[Int] =
  (rx, ry) match {
    case (Value(x), Value(y)) => Value(op(x, y))
    case (Error(m), _)        => Error(m)
    case (_, Error(m))        => Error(m)
  }
```

Now we can easily "map" any binary operation on integers to a binary operation on `Result[Int]`, assuming that the operation never generates an error:

```
def sub(rx: Result[Int], ry: Result[Int]): Result[Int] = map2(rx, ry){ (x, y) => x - y }
```

Custom code is still needed for operations that *may* generate errors:

```
def div(rx: Result[Int], ry: Result[Int]): Result[Int] = (rx, ry) match {
  case (Value(x), Value(y)) if y != 0  => Value(x / y)
  case (Value(x), Value(y))            => Error(s"error: $x / $y")
  case (Error(m), _)                   => Error(m)
  case (_, Error(m))                   => Error(m)
}
```

We can now test the "safe arithmetic" on simple calculations:

```
scala> add(Value(1), Value(2))
res10: Result[Int] = Value(3)

scala> div(add(Value(1), Value(2)), Value(0))
res11: Result[Int] = Error(error: 3 / 0)
```

We see that indeed all further computations are abandoned once an error occurs. An error message shows only the immediate calculation that generated the error. For instance, the error message for $20 + 1/0$ never mentions 20:

```
scala> add(Value(20), div(Value(1), Value(0)))
res12: Result[Int] = Error(error: 1 / 0)

scala> add(sqrt(Value(-1)), Value(10))
res13: Result[Int] = Error(error: sqrt(-1))
```

3.2.3 Standard disjunctive types: `Option`, `Either`, `Try`

The Scala library defines the disjunctive types `Option`, `Either`, and `Try` because they are used often. We now look at each of them in turn.

The `Option` type is a disjunctive type with two cases: the empty tuple and a one-element tuple. The names of the two case classes are `None` and `Some`. If the `Option` type were not already defined in the standard library, one could define it with the code

```
sealed trait Option[T]
final case object None          extends Option[Nothing]
final case class Some[T](t: T)  extends Option[T]
```

This code is similar to the type `SearchResult` defined in Section 3.2.1, except that `Option` has a type parameter instead of a fixed type `Int`. Another difference is the use of a `case object` for the empty case instead of an empty case class, such as `None()`. Since Scala's `case object`s cannot have type parameters, the type parameter in the definition of `None` must be set to the special type `Nothing`, which is a type with *no* values (also called the **void type**).

An alternative (implemented in libraries such as `scalaz`) is to define the empty option value as

```
final case class None[T]() extends Option[T]
```

In that implementation, the empty option `None[T]()` has a type parameter.

Several consequences follow from the Scala library's decision to define `None` without a type parameter. One consequence is that `None` can be reused as a value of type `Option[A]` for any type `A`:

```
scala> val y: Option[Int] = None
y: Option[Int] = None

scala> val z: Option[String] = None
z: Option[String] = None
```

Typically, `Option` is used in situations where a value may be either present or missing, especially when a missing value is *not an error*. The missing-value case is represented by `None`, while `Some(x)` means that a value `x` is present.

Example 3.2.3.1 Information about "subscribers" must include a name and an email address, but a telephone number is optional. To represent this information, we define a case class like this,

```
case class Subscriber(name: String, email: String, phone: Option[Long])
```

What if we represent the missing telephone number by a special value such as `-1` and use the simpler type `Long` instead of `Option[Long]`? The disadvantage is that we would need to *remember* to check for the special value `-1` in all functions that take the telephone number as an argument. Looking at a function such as `sendSMS(phone: Long)` at a different place in the code, a programmer might forget that the telephone number is actually optional. In contrast, the type signature `sendSMS(phone: Option[Long])` unambiguously indicates that the telephone number might be missing and helps the programmer to remember to handle both cases.

Pattern-matching code involving `Option` can handle the two cases like this:

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone match {
  case None              => None      // Cannot obtain digits, so return `None`.
  case Some(number)      => Some(digitsOf(number))
}  // The function `digitsOf` was defined in Section 2.3.
```

At the two sides of "`case None => None`", the value `None` has different types, namely `Option[Long]` and `Option[Seq[Long]]`. Since these types are declared in the type signature of the function `getDigits`, the Scala compiler is able to figure out the types of all expressions in the `match`/`case` construction. So, pattern-matching code can be written without explicit type annotations such as `(None: Option[Long])`.

If we now need to compute the number of digits, we can write

```
def numberOfDigits(phone: Option[Long]): Option[Long] = getDigits(phone) match {
  case None              => None
  case Some(digits)      => Some(digits.length)
}
```

These examples perform a computation when an `Option` value is non-empty, and leave it empty otherwise. This code pattern is used often. To avoid repeating the code, we can implement this code pattern as a function that takes the computation as an argument `f`:

```
def doComputation(x: Option[Long], f: Long => Long): Option[Long] = x match {
  case None              => None
  case Some(i)           => Some(f(i))
}
```

It is then natural to generalize this function to arbitrary types using type parameters instead of a fixed type `Long`. The resulting function is usually called `fmap` in functional programming libraries:

```
def fmap[A, B](f: A => B): Option[A] => Option[B] = {
  case None              => None
  case Some(a)           => Some(f(a))
}

scala> fmap(digitsOf)(Some(4096))
```

```
res0: Option[Seq[Long]] = Some(List(4, 0, 9, 6))

scala> fmap(digitsOf)(None)
res1: Option[Seq[Long]] = None
```

We say that the `fmap` operation **lifts** a given function of type `A => B` to the type `Option[A] => Option[B]`.

It is important to keep in mind that the code `case Some(a) => Some(f(a))` changes the type of the option value. On the left side of the arrow, the type is `Option[A]`, while on the right side it is `Option[B]`. The Scala compiler knows this from the given type signature of `fmap`, so an explicit type parameter, which we could write as `Some[B](f(a))`, is not needed.

The Scala library implements an equivalent function as a method of the `Option` class, with the syntax `x.map(f)` rather than `fmap(f)(x)`. We can concisely rewrite the previous code using these methods,

```
def getDigits(phone: Option[Long]): Option[Seq[Long]] = phone.map(digitsOf)
def numberOfDigits(phone: Option[Long]): Option[Long] = phone.map(digitsOf).map(_.length)
```

We see that the `map` operation for the `Option` type is analogous to the `map` operation for sequences.

The similarity between `Option[A]` and `Seq[A]` is clearer if we view `Option[A]` as a special kind of "sequence" whose length is restricted to be either 0 or 1. So, `Option[A]` can have all the operations of `Seq[A]` except operations such as `concat` that may grow the sequence beyond length 1. The standard operations defined on `Option` include `map`, `filter`, `zip`, `forall`, `exists`, `flatMap`, and `foldLeft`.

Example 3.2.3.2 Given a phone number as `Option[Long]`, extract the country code if it is present. (Assume that the country code is any digits in front of the 10-digit number; for the phone number 18004151212, the country code is 1.) The result must be again of type `Option[Long]`.

Solution If the phone number is a positive integer n, we may compute the country code simply as `n/10000000000L`. However, if the result of that division is zero, we should return an empty `Option` (i.e. the value `None`) rather than `0`. To implement this logic, we may begin by writing this code,

```
def countryCode(phone: Option[Long]): Option[Long] = phone match {
  case None      => None
  case Some(n)   =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}
```

We may notice that we have reimplemented the code pattern similar to `map` in this code, namely "if `None` then return `None`, else do a computation". So we may try to rewrite the code as

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Type error: the result is Option[Option[Long]], not Option[Long].
```

This code does not compile: we are returning an `Option[Long]` within a function lifted via `map`, so the resulting type is `Option[Option[Long]]`. We may use `flatten` to convert `Option[Option[Long]]` to the required type `Option[Long]`,

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
}.flatten // Types are correct now.
```

Since the `flatten` follows a `map`, we can rewrite the code using `flatMap`:

```
def countryCode(phone: Option[Long]): Option[Long] = phone.flatMap { n =>
    val countryCode = n / 10000000000L
    if (countryCode != 0L) Some(countryCode) else None
} // Types are correct now.
```

Another way of implementing this example is to notice the code pattern "if condition does not hold, return `None`, otherwise keep the value". For an `Option` type, this is equivalent to the `filter` operation (recall that `filter` returns an empty sequence if the predicate never holds). The code is

```
def countryCode(phone: Option[Long]): Option[Long] = phone.map(_ / 10000000000L).filter(_ != 0L)

scala> countryCode(Some(18004151212L))
res0: Option[Long] = Some(1)

scala> countryCode(Some(8004151212L))
res1: Option[Long] = None
```

Example 3.2.3.3 Add a new requirement to Example 3.2.3.2: if the country code is not present, we should return the default country code 1.

Solution This is an often used code pattern: "if empty, substitute a default value". The Scala library has the method `getOrElse` for this purpose:

```
scala> Some(100).getOrElse(1)
res2: Int = 100

scala> None.getOrElse(1)
res3: Int = 1
```

So we can implement the new requirement as

```
scala> countryCode(Some(8004151212L)).getOrElse(1L)
res4: Long = 1
```

Using `Option` with collections Many Scala library methods return an `Option` as a result. The main examples are `find`, `headOption`, and `lift` for sequences, and `get` for dictionaries.

The `find` method returns the first element satisfying a predicate:

```
scala> (1 to 10).find(_ > 5)
res0: Option[Int] = Some(6)

scala> (1 to 10).find(_ > 10) // No element is > 10.
res1: Option[Int] = None
```

The `lift` method returns the element of a sequence at a given index:

```
scala> (10 to 100).lift(0)
res2: Option[Int] = Some(10)

scala> (10 to 100).lift(1000) // No element at index 1000.
res3: Option[Int] = None
```

The `headOption` method returns the first element of a sequence, unless the sequence is empty. This is equivalent to `lift(0)`:

```
scala> Seq(1,2,3).headOption
res4: Option[Int] = Some(1)

scala> Seq(1,2,3).filter(_ > 10).headOption
res5: Option[Int] = None
```

Applying `.find(p)` computes the same result as `.filter(p).headOption`, but `.find(p)` may be faster.

The `get` method for a dictionary returns the value if it exists for a given key, and returns `None` if the key is not in the dictionary:

```
scala> Map(10 -> "a", 20 -> "b").get(10)
res6: Option[String] = Some(a)

scala> Map(10 -> "a", 20 -> "b").get(30)
res7: Option[String] = None
```

The `get` method is a safe by-key access to dictionaries, unlike the direct access that may fail:

```
scala> Map(10 -> "a", 20 -> "b")(10)
res8: String = a
```

```
scala> Map(10 -> "a", 20 -> "b")(30)
java.util.NoSuchElementException: key not found: 30
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  ... 32 elided
```

Similarly, `lift` is a safe by-index access to collections, unlike the direct access that may fail:

```
scala> Seq(10,20,30)(0)
res9: Int = 10

scala> Seq(10,20,30)(5)
java.lang.IndexOutOfBoundsException: 5
  at scala.collection.LinearSeqOptimized$class.apply(LinearSeqOptimized.scala:65)
  at scala.collection.immutable.List.apply(List.scala:84)
  ... 32 elided
```

The `Either` type The standard disjunctive type `Either[A, B]` has two type parameters and is often used for computations that report errors. By convention, the *first* type (`A`) is the type of error, and the *second* type (`B`) is the type of the (non-error) result. The names of the two cases are `Left` and `Right`. A possible definition of `Either` may be written as

```
sealed trait Either[A, B]
final case class  Left[A, B](value: A) extends Either[A, B]
final case class Right[A, B](value: B) extends Either[A, B]
```

By convention, a value `Left(x)` represents an error, and a value `Right(y)` represents a valid result.

As an example, the following function substitutes a default value and logs the error information:

```
def logError(x: Either[String, Int], default: Int): Int = x match {
  case Left(error)  => println(s"Got error: $error"); default
  case Right(res)   => res
}
```

To test:

```
scala> logError(Right(123), -1)
res1: Int = 123

scala> logError(Left("bad result"), -1)
Got error: bad result
res2: Int = -1
```

Why use `Either` instead of `Option` for computations that may fail? A failing computation such as `"xyz".toInt` cannot return a result, and sometimes we might use `None` to indicate that a result is not available. However, when the result is a requirement for further calculations, we will usually need to know exactly *which* error prevented the result from being available. The `Either` type may provide detailed information about such errors, which `Option` cannot do.

The `Either` type generalizes the type `Result` defined in Section 3.2.1 to an arbitrary error type instead of `String`. We have seen its usage in Example 3.2.2.4, where the code pattern was "if value is present, do a computation, otherwise keep the error". This code pattern is implemented by the `map` method of `Either`:

```
1 scala> Right(1).map(_ + 1)
2 res0: Either[Nothing, Int] = Right(2)
3
4 scala> Left[String, Int]("error").map(_ + 1)
5 res1: Either[String, Int] = Left("error")
```

The type `Nothing` was filled in by the Scala compiler because we did not specify the first type parameter of `Right` in line 1.

The methods `flatMap`, `fold`, and `getOrElse` are also defined for `Either`, with the same convention that a `Left` value represents an error.[1]

Exceptions and the `Try` type When computations fail for any reason, Scala generates an **exception** instead of returning a value. An exception means that the evaluation of some expression was stopped without returning a result.

As an example, exceptions are generated when the available memory is too small to store the resulting data (as we saw in Section 2.6.3), or if a stack overflow occurs during the computation (as we saw in Section 2.2.3). Exceptions may also occur due to programmer's error: when a pattern matching operation fails, when a requested key does not exist in a dictionary, or when the `head` operation is applied to an empty list.

Motivated by these examples, we may distinguish "planned" and "unplanned" exceptions.

A **planned** exception is generated by programmer's code via the `throw` syntax:

```
scala> throw new Exception("This is a test... this is only a test.")
java.lang.Exception: This is a test... this is only a test.
```

The Scala library contains a `throw` operation in various places, such as in the code for applying the `head` method to an empty sequence, as well as in other situations where exceptions are generated due to programmer's errors. These exceptions are generated deliberately and in well-defined situations. Although these exceptions indicate errors, these errors are anticipated in advance and so may be handled by the programmer.

For example, many Java libraries will generate exceptions when function arguments have unexpected values, when a network operation takes too long or a network connection is unexpectedly broken, when a file is not found or cannot be read due to access permissions, and in many other situations. All these exceptions are "planned" because they are generated explicitly by library code such as `throw new FileNotFoundException(...)`. The programmer's code is expected to catch these exceptions, to handle the error, and to continue the evaluation of the program.

An **unplanned** exception is generated by the Java runtime system when critical errors occur, such as an out-of-memory error. It is rare that a programmer writes `val y = f(x)` while *expecting* that an out-of-memory exception will sometimes occur at that point.[2] An unplanned exception indicates a serious and unforeseen problem with memory or another critically important resource, such as the operating system's threads or file handles. Such problems usually cannot be fixed and will prevent the program from running any further. It is reasonable that the program should abruptly stop (or "crash" as programmers say) after such an error.

The use of planned exceptions assumes that the programmer will write code to handle each exception. This assumption makes it significantly harder to write programs correctly: it is hard to figure out and to keep in mind all the possible exceptions that a given library function may `throw` in its code (and in the code of all other libraries on which it depends). Instead of using exceptions for indicating errors, Scala programmers can write functions that return a disjunctive type, such as `Either`, describing both a correct result and an error condition. Users of these functions will *have* to do pattern matching on the result values. This helps programmers to remember and to handle all relevant error situations that the programmers anticipate to encounter.

However, programmers will often need to use Java or Scala libraries that `throw` exceptions. To help write code for these situations, the Scala library contains a helper function called `Try()` and a disjunctive type, also called `Try`. The type `Try[A]` is equivalent to `Either[Throwable, A]`, where `Throwable` is the general type of all exceptions (i.e. values to which a `throw` operation can be applied). The two parts of the disjunctive type `Try[A]` are called `Failure` and `Success[A]` (instead of `Left[Throwable, A]` and `Right[Throwable, A]` in the `Either` type). The function `Try(expr)` will catch all exceptions thrown while the expression `expr` is evaluated. If the evaluation of `expr` succeeds and returns a value `x: A`, the value of `Try(expr)` will be `Success(x)`. Otherwise it will be `Failure(t)`, where `t: Throwable` is the value associated with the generated exception. Here is an example of using `Try`:

```
import scala.util.{Try, Success, Failure}
```

[1] These methods are available in Scala 2.12 or a later version.

[2] Just once in the author's experience, an out-of-memory condition had to be anticipated in an Android app.

```
scala> val p = Try("xyz".toInt)
p: Try[Int] = Failure(java.lang.NumberFormatException: For input string: "xyz")

scala> val q = Try("0002".toInt)
q: Try[Int] = Success(2)
```

The code `Try("xyz".toInt)` does not generate any exceptions and will not crash the program. Any computation that may `throw` an exception can be enclosed in a `Try()`, and the exception will be caught and encapsulated within the disjunctive type as a `Failure(...)` value.

The methods `map`, `filter`, `flatMap`, `foldLeft` are defined for the `Try` class similarly to the `Either` type. One additional feature of `Try` is to catch exceptions generated by the function arguments of `map`, `filter`, `flatMap`, and other standard methods:

```
scala> val y = q.map(y => throw new Exception("ouch"))
y: Try[Int] = Failure(java.lang.Exception: ouch)

scala> val z = q.filter(y => throw new Exception("huh"))
z: Try[Int] = Failure(java.lang.Exception: huh)
```

In this example, the values `y` and `z` were computed *successfully* even though exceptions were thrown while the function arguments of `map` and `filter` were evaluated. Further code can use pattern matching on the values `y` and `z` and examine those exceptions. However, it is important that these exceptions were caught and the program did not crash, meaning that further code is *able* to run.

While the standard types `Try` and `Either` will cover many use cases, programmers can also define custom disjunctive types in order to represent all the anticipated failures or errors in the business logic of a particular application. Representing all errors in the types helps assure that the program will not crash because of an exception that we forgot to handle or did not even know about.

3.3 Lists and trees: recursive disjunctive types

Consider this code defining a disjunctive type `NInt`:

```
sealed trait NInt
final case class N1(x: Int)    extends NInt
final case class N2(n: NInt)   extends NInt
```

The type `NInt` has two disjunctive parts, `N1` and `N2`. But the definition of the case class `N1` refers to the type `NInt` as if it were already defined.

A type whose definition uses that same type is called a **recursive type**. The type `NInt` is an example of a recursive disjunctive type.

We might imagine defining a disjunctive type `X` whose parts recursively refer to the same type `X` (and/or to each other) in complicated ways. What would kind of data would be represented by such a type `X`, and in what situations would `X` be useful? For instance, the simple definition

```
final case class Bad(x: Bad)
```

is useless: we can create a value of type `Bad` only if we already have a value of type `Bad`. This is an example of an infinite loop in type recursion. We will never be able to create any values of type `Bad`, which means that the type `Bad` is void (has no values, like the special type `Nothing`).

Section 8.5.1 will derive conditions for recursive types to be non-void. For now, we will look at the recursive disjunctive types that are used most often: lists and trees.

3.3.1 Lists

A list of values of type `A` is either empty, or has one value of type `A`, or two values of type `A`, etc. We can visualize the type `List[A]` as a disjunctive type defined by

```
sealed trait List[A]
final case class List0[A]()               extends List[A]
final case class List1[A](x: A)           extends List[A]
final case class List2[A](x1: A, x2: A)   extends List[A]
???                // Need an infinitely long definition.
```

However, this definition is not practical: we cannot define a separate case class for *each* possible length. Instead, we define the type `List[A]` via mathematical induction on the length of the list:

- Base case: empty list, `case class List0[A]()`.
- Inductive step: given a list of a previously defined length, say $\texttt{List}_{n-1}$, define a new case class $\texttt{List}_n$ describing a list with one more element of type `A`. So we could define $\texttt{List}_n$ = `(A,` $\texttt{List}_{n-1}$`)`.

Let us try to write this inductive definition as code:

```
sealed trait ListI[A]          // Inductive definition of a list.
final case class List0[A]()                       extends ListI[A]
final case class List1[A](x: A, next: List0[A])   extends ListI[A]
final case class List2[A](x: A, next: List1[A])   extends ListI[A]
???                   // Still need an infinitely long definition.
```

To avoid writing an infinitely long type definition, we need to use a trick. Notice that all definitions of `List1`, `List2`, etc., have a similar form (while `List0` is not similar). We can replace all the definitions `List1`, `List2`, etc., by a single definition if we use the type `ListI[A]` recursively inside the case class:

```
sealed trait ListI[A]          // Inductive definition of a list.
final case class List0[A]()                       extends ListI[A]
final case class ListN[A](x: A, next: ListI[A])   extends ListI[A]
```

The type definition has become recursive. For this trick to work, it is important to use `ListI[A]` and not `ListN[A]` inside the definition `ListN[A]`. Otherwise, we would get an infinite loop in type recursion, similarly to `case class Bad` shown before.

Since we obtained the type definition of `ListI` via a trick, let us verify that the code actually defines the disjunctive type we wanted.

To create a value of type `ListI[A]`, we must use one of the two available case classes. Using the first case class, we may create a value `List0()`. Since this empty case class does not contain any values of type `A`, it effectively represents an empty list (the base case of the induction). Using the second case class, we may create a value `ListN(x, next)` where `x` is of type `A` and `next` is an already constructed value of type `ListI[A]`. This represents the inductive step because the case class `ListN` is a named tuple containing `A` and `ListI[A]`. Now, the same consideration recursively applies to constructing the value `next`, which must be either an empty list or a pair containing a value of type `A` and another list. The assumption that the value `next: ListI[A]` is already constructed is equivalent to the inductive assumption that we already have a list of a previously defined length. So, we have verified that `ListI[A]` implements the inductive definition shown above.

Examples of values of type `ListI` are the empty list `List0()`, a one-element list `ListN(x, List0())`, and a two-element list `ListN(x, ListN(y, List0()))`.

To illustrate writing pattern-matching code using this type, let us implement the method `headOption`:

```
def headOption[A]: ListI[A] => Option[A] = {
  case List0()          => None
  case ListN(x, next)   => Some(x)
}
```

The Scala library already defines the type `List[A]` in a different but equivalent way:

```
sealed trait List[A]
final case object Nil extends List[Nothing]
final case class ::[A](head: A, tail: List[A]) extends List[A]
```

Because "operator-like" case class names, such as `::`, support the infix syntax, we may write expressions such as `head :: tail` instead of `::(head, tail)`. This syntax can be also used in pattern matching on `List` values, with code that looks like this:

```
def headOption[A]: List[A] => Option[A] = {
  case Nil            => None
  case head :: tail   => Some(head)
}
```

Examples of values created using Scala's standard `List` type are the empty list `Nil`, a one-element list `x :: Nil`, and a two-element list `x :: y :: Nil`. The same syntax `x :: y :: Nil` is used both for creating values of type `List` and for pattern-matching on such values.

The Scala library also defines the helper function `List()`, so that `List()` is the same as `Nil` and `List(1, 2, 3)` is the same as `1 :: 2 :: 3 :: Nil`. Lists are easier to use in the syntax `List(1, 2, 3)`. Pattern matching can also use that syntax when convenient:

```
val x: List[Int] = List(1, 2, 3)

x match {
  case List(a)          => ...
  case List(a, b, c)    => ...
  case _                => ...
}
```

3.3.2 Tail recursion with `List`

Because the `List` type is defined by induction, it is straightforward to implement iterative computations with the `List` type using recursion.

A first example is the `map` function. We use reasoning by induction in order to figure out the implementation of `map`. The required type signature is

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

The base case is an empty list, and we return again an empty list:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil => Nil
  ...
```

In the inductive step, we have a pair `(head, tail)` in the case class `::`, with `head:A` and `tail:List[A]`. The pair can be pattern-matched with the syntax `head :: tail`. The `map` function should apply the argument `f` to the head value, which will give the first element of the resulting list. The remaining elements are computed by the induction assumption, i.e. by a recursive call to `map`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] = xs match {
  case Nil           => Nil
  case head :: tail  => f(head) :: map(tail)(f) // Not tail-recursive.
```

While this implementation is straightforward and concise, it is not tail-recursive. This will be a problem for large enough lists.

Instead of implementing the often-used methods such as `map` or `filter` one by one, let us implement `foldLeft`, because most of the other methods can be expressed via `foldLeft`.

The required type signature is

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = ???
```

Reasoning by induction, we start with the base case `xs == Nil`, where the only possibility is to return the value `init`:

```
def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R = xs match {
    case Nil             => init
    ...
```

The inductive step for `foldLeft` says that, given the values `head:A` and `tail:List[A]`, we need to apply the updater function to the previous accumulator value. That value is `init`. So we apply `foldLeft` recursively to the tail of the list once we have the updated accumulator value:

```
@tailrec def foldLeft[A, R](xs: List[A])(init: R)(f: (R, A) => R): R =
  xs match {
    case Nil             => init
    case head :: tail    =>
```

```
      val newInit = f(init, head) // Update the accumulator.
      foldLeft(tail)(newInit)(f)  // Recursive call to `foldLeft`.
  }
```

This implementation is tail-recursive because the recursive call to `foldLeft` is the last expression returned in a `case` branch.

Another example is a function for reversing a list. The Scala library defines the `reverse` method for this task, but we will show an implementation using `foldLeft`. The updater function *prepends* an element to a previous list:

```
def reverse[A](xs: List[A]): List[A] =
  xs.foldLeft(Nil: List[A])((prev, x) => x :: prev)

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

Without the explicit type annotation `Nil:List[A]`, the Scala compiler will decide that `Nil` has type `List[Nothing]`, and the types will not match later in the code. In Scala, one often finds that the initial value for `foldLeft` needs an explicit type annotation.

The `reverse` function can be used to obtain a tail-recursive implementation of `map` for `List`. The idea is to first use `foldLeft` to accumulate transformed elements:

```
scala> Seq(1, 2, 3).foldLeft(Nil:List[Int])((prev, x) => x*x :: prev)
res0: List[Int] = List(9, 4, 1)
```

The result is a reversed `.map(x => x*x)`, so we need to apply `reverse`:

```
def map[A, B](xs: List[A])(f: A => B): List[B] =
  xs.foldLeft(Nil: List[B])((prev, x) => f(x) :: prev).reverse

scala> map(List(1, 2, 3))(x => x*x)
res2: List[Int] = List(1, 4, 9)
```

This achieves stack safety at the cost of traversing the list twice. (This code is shown only as an example. The Scala library implements `map` using low-level tricks for better performance.)

Example 3.3.2.1 A definition of the **non-empty list** is similar to `List` except that the empty-list case is replaced by a 1-element case:

```
sealed trait NEL[A]
final case class Last[A](head: A)                  extends NEL[A]
final case class More[A](head: A, tail: NEL[A])    extends NEL[A]
```

Values of a non-empty list look like this:

```
scala> val xs: NEL[Int] = More(1, More(2, Last(3)))  // [1, 2, 3]
xs: NEL[Int] = More(1,More(2,Last(3)))

scala> val ys: NEL[String] = Last("abc") // One element, ["abc"].
ys: NEL[String] = Last(abc)
```

To create non-empty lists more easily, we implement a conversion function `toNEL` from an ordinary list. To guarantee that a non-empty list can be created, we give `toNEL` *two* arguments:

```
def toNEL[A](x: A, rest: List[A]): NEL[A] = rest match {
  case Nil          => Last(x)
  case y :: tail    => More(x, toNEL(y, tail))
}  // Not tail-recursive: `toNEL()` is used inside `More(...)`.
```

To test:

```
scala> toNEL(1, List()) // Result = [1].
res0: NEL[Int] = Last(1)

scala> toNEL(1, List(2, 3)) // Result = [1, 2, 3].
res1: NEL[Int] = More(1,More(2,Last(3)))
```

The `head` method is safe for non-empty lists, unlike `head` for an ordinary `List`:

```
def head[A]: NEL[A] => A = {
    case Last(x)        => x
    case More(x, _)     => x
}
```

We can also implement a tail-recursive `foldLeft` function for non-empty lists:

```
@tailrec def foldLeft[A, R](n: NEL[A])(init: R)(f: (R, A) => R): R = n match {
    case Last(x)        => f(init, x)
    case More(x, tail)  => foldLeft(tail)(f(init, x))(f)
  }

scala> foldLeft(More(1, More(2, Last(3))))(0)(_ + _)
res2: Int = 6
```

Example 3.3.2.2 Use `foldLeft` to implement a `reverse` function for the type `NEL`. The required type signature and a sample test:

```
def reverse[A]: NEL[A] => NEL[A] = ???

scala> reverse(toNEL(10, List(20, 30))) // Result must be [30, 20, 10].
res3: NEL[Int] = More(30,More(20,Last(10)))
```

Solution We will use `foldLeft` to build up the reversed list as the accumulator value. It remains to choose the initial value of the accumulator and the updater function. We have already seen the code for reversing the ordinary list via the `foldLeft` method (Section 3.3.2),

```
def reverse[A](xs: List[A]): List[A] = xs.foldLeft(Nil: List[A])((prev,x) => x :: prev)
```

However, we cannot reuse the same code for non-empty lists by writing `More(x, prev)` instead of `x :: prev`, because the `foldLeft` operation works with non-empty lists differently. Since lists are always non-empty, the updater function is always applied to an initial value, and the code works incorrectly:

```
def reverse[A](xs: NEL[A]): NEL[A] =
  foldLeft(xs)(Last(head(xs)):NEL[A])((prev,x) => More(x, prev))

scala> reverse(toNEL(10,List(20,30))) // Result = [30, 20, 10, 10].
res4: NEL[Int] = More(30,More(20,More(10,Last(10))))
```

The last element, `10`, should not have been repeated. It was repeated because the initial accumulator value already contained the head element `10` of the original list. However, we cannot set the initial accumulator value to an empty list, since a value of type `NEL[A]` must be non-empty. It seems that we need to handle the case of a one-element list separately. So we begin by matching on the argument of `reverse`, and apply `foldLeft` only when the list is longer than 1 element:

```
def reverse[A]: NEL[A] => NEL[A] = {
    case Last(x)         => Last(x)    // ‘reverse‘ is trivial.
    case More(x, tail)   =>            // Use foldLeft on ‘tail‘.
      foldLeft(tail)(Last(x):NEL[A])((prev,x) => More(x, prev))
  }

scala> reverse(toNEL(10, List(20, 30))) // Result = [30, 20, 10].
res5: NEL[Int] = More(30,More(20,Last(10)))
```

Exercise 3.3.2.3 Implement a function `toList` that converts a non-empty list into an ordinary Scala `List`. The required type signature and a sample test:

```
def toList[A](nel: NEL[A]): List[A] = ???

scala> toList(More(1, More(2, Last(3)))) // This is [1, 2, 3].
res6: List[Int] = List(1, 2, 3)
```

Exercise 3.3.2.4 Implement a `map` function for the type `NEL`. Type signature and a sample test:

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

scala> map[Int, Int](toNEL(10, List(20, 30)))(_ + 5) // Result = [15, 25, 35].
res7: NEL[Int] = More(15,More(25,Last(35)))
```

Exercise 3.3.2.5 Implement a function `concat` that concatenates two non-empty lists:

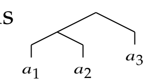
```
def concat[A](xs: NEL[A], ys: NEL[A]): NEL[A] = ???

scala> concat(More(1, More(2, Last(3))), More(4, Last(5))) // Result is [1, 2, 3, 4, 5].
res8: NEL[Int] = More(1,More(2,More(3,More(4,Last(5)))))
```

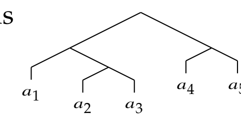
3.3.3 Binary trees

We will consider four kinds of trees defined as recursive disjunctive types: binary trees, rose trees, regular-shaped trees, and abstract syntax trees.

Examples of a **binary tree** with leaves of type `A` can be drawn as



or as



,

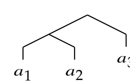
where a_i are some values of type `A`.

An inductive definition says that a binary tree is either a leaf with a value of type `A` or a branch containing *two* previously defined binary trees. Translating this definition into code, we get

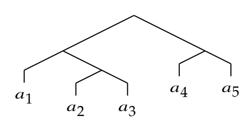
```
sealed trait Tree2[A]
final case class Leaf[A](a: A)                        extends Tree2[A]
final case class Branch[A](x: Tree2[A], y: Tree2[A])   extends Tree2[A]
```

Here are some examples of code expressions and the corresponding trees that use this definition:

```
Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))
```




```
Branch(Branch(Leaf("a1"), Branch(Leaf("a2"), Leaf("a3"))),
  Branch(Leaf("a4"), Leaf("a5")))
```



Recursive functions on trees are translated into concise code. For instance, the function `foldLeft` for trees of type `Tree2` is implemented as

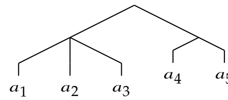
```
def foldLeft[A, R](t: Tree2[A])(init: R)(f: (R, A) => R): R = t match {
  case Leaf(a)          => f(init, a)
  case Branch(t1, t2)   =>
    val r1 = foldLeft(t1)(init)(f) // Fold the left branch.
    foldLeft(t2)(r1)(f) // Starting from ‘r1‘, fold the right branch.
}
```

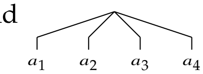
Note that this function *cannot* be made tail-recursive using the accumulator trick, because `foldLeft` needs to call itself twice in the `Branch` case. To test:

```
val t: Tree2[String] = Branch(Branch(Leaf("a1"), Leaf("a2")), Leaf("a3"))

scala> foldLeft(t)("")(_ + " " + _)
res0: String = " a1 a2 a3"
```

3.3.4 Rose trees

A **rose tree** is similar to the binary tree except the branches contain a non-empty list of trees. Because of that, a rose tree can fork into arbitrarily many branches at each node, rather than always into two branches as the binary tree does; for example, 

and 

are rose trees.

A possible definition of a data type for the rose tree is

```
sealed trait TreeN[A]
final case class Leaf[A](a: A)                      extends TreeN[A]
final case class Branch[A](ts: NEL[TreeN[A]])   extends TreeN[A]
```

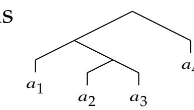
Since we used a non-empty list `NEL`, a `Branch()` value is guaranteed to have at least one branch. If we used an ordinary `List` instead, we could (by mistake) create a tree with no leaves and no branches.

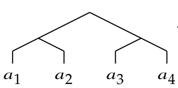
Exercise 3.3.4.1 Define the function `foldLeft` for a rose tree, using `foldLeft` for the type `NEL`. Type signature and a test:

```
def foldLeft[A, R](t: TreeN[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(More(Leaf(1), More(Leaf(2), Last(Leaf(3))))))(0)(_ + _)
res0: Int = 6
```

3.3.5 Regular-shaped trees

Binary trees and rose trees may choose to branch or not to branch at any given node, resulting in structures that may have different branching depths at different nodes, such as 

. A **regular-shaped tree** always branches in the same way at every node until a chosen total depth, e.g., 

, where all nodes at depth 0 and 1 always branch into two, while nodes at depth 2 do not branch. The branching number is fixed for a given type of a regular-shaped tree; in this example, the branching number is 2, so it is a regular-shaped *binary* tree.

How can we define a data type representing a regular-shaped binary tree? We need a type that is either a single value, or a pair of values, or a pair of pairs, etc. Begin with the non-recursive (but, of course, impractical) definition

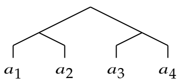
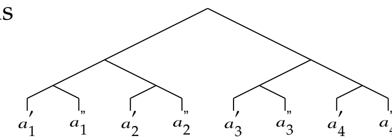
```
sealed trait RTree[A]
final case class Leaf[A](x: A)                          extends RTree[A]
final case class Branch1[A](xs: (A, A))             extends RTree[A]
final case class Branch2[A](xs: ((A, A),(A, A)))   extends RTree[A]
???                          // Need an infinitely long definition.
```

The case `Branch1` describes a regular-shaped tree with total depth 1, the case `Branch2` has total depth 2, and so on. Now, we cannot rewrite this definition as a recursive type because the case classes do not have the same structure. The non-trivial trick is to notice that each case class `Branch`$_n$ uses the previous case class's data structure *with the type parameter* set to `(A, A)` instead of `A`. So we can rewrite this definition as

```
sealed trait RTree[A]
final case class Leaf[A](x: A)                          extends RTree[A]
final case class Branch1[A](xs: Leaf[(A, A)])      extends RTree[A]
final case class Branch2[A](xs: Branch1[(A, A)])   extends RTree[A]
???                          // Need an infinitely long definition.
```

We can now apply the type recursion trick: replace the type `Branch`$_{n-1}$`[(A, A)]` in the definition of `Branch`$_n$ by the recursively used type `RTree[(A, A)]`. Now we can define a regular-shaped binary tree:

```
sealed trait RTree[A]
final case class Leaf[A](x: A)                      extends RTree[A]
final case class Branch[A](xs: RTree[(A, A)])   extends RTree[A]
```

Since we used some tricks to figure out the definition of `RTree[A]`, let us verify that this definition actually describes the recursive disjunctive type we wanted. The only way to create a structure of type `RTree[A]` is to create a `Leaf[A]` or a `Branch[A]`. A value of type `Leaf[A]` is a correct regularly-shaped tree. It remains to consider the case of `Branch[A]`. Creating a `Branch[A]` requires a previously created `RTree` with values of type `(A, A)` instead of `A`. By the inductive assumption, the previously created `RTree[A]` would have the correct shape. Now, it is clear that if we replace the type parameter `A` by the pair `(A, A)`, a regular-shaped tree such as  remains regular-shaped but becomes one level deeper, which can be drawn (replacing each a_i by a pair a_i', a_i'') as .

We see that `RTree[A]` is a correct definition of a regular-shaped binary tree.

Example 3.3.5.1 Define a (non-tail-recursive) `map` function for a regular-shaped binary tree. The required type signature and a test:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = ???

scala> map(Branch(Branch(Leaf(((1,2),(3,4))))))(_ * 10)
res0: RTree[Int] = Branch(Branch(Leaf(((10,20),(30,40)))))
```

Solution Begin by pattern-matching on the tree:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)       => ???
  case Branch(xs)    => ???
}
```

In the base case, we have no choice but to return `Leaf(f(x))`.

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)       => Leaf(f(x))
  case Branch(xs)    => ???
}
```

In the inductive step, we are given a previous tree value `xs:RTree[(A, A)]`. It is clear that we need to apply `map` recursively to `xs`. Let us try:

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)       => Leaf(f(x))
  case Branch(xs)    => Branch(map(xs)(f))    // Type error!
}
```

Here, `map(xs)(f)` has an incorrect type of the function `f`. Since `xs` has type `RTree[(A, A)]`, the recursive call `map(xs)(f)` requires `f` to be of type `((A, A)) => (B, B)` instead of `A => B`.

So, we need to provide a function of the correct type instead of `f`. A function of type `((A, A)) => (B, B)` will be obtained out of `f: A => B` if we apply `f` to each part of the tuple `(A, A)`; the code for that function is `{ case (x, y) => (f(x), f(y)) }`. Therefore, we can implement `map` as

```
def map[A, B](t: RTree[A])(f: A => B): RTree[B] = t match {
  case Leaf(x)       => Leaf(f(x))
  case Branch(xs)    => Branch(map(xs){ case (x, y) => (f(x), f(y)) })
}
```

This code is not tail-recursive since it calls `map` inside an expression.

Exercise 3.3.5.2 Using tail recursion, compute the depth of a regular-shaped binary tree of type `RTree`. (An `RTree` of depth n has 2^n leaf values.) The required type signature and a test:

```
@tailrec def depth[A](t: RTree[A]): Int = ???

scala> depth(Branch(Branch(Leaf((("a","b"),("c","d"))))))
res2: Int = 2
```

Exercise 3.3.5.3* Define a tail-recursive function `foldLeft` for a regular-shaped binary tree. The required type signature and a test:

```
@tailrec def foldLeft[A, R](t: RTree[A])(init: R)(f: (R, A) => R): R = ???

scala> foldLeft(Branch(Branch(Leaf(((1,2),(3,4))))))(0)(_ + _)
res0: Int = 10

scala> foldLeft(Branch(Branch(Leaf((("a","b"),("c","d"))))))("")(_ + _)
res1: String = abcd
```

3.3.6 Abstract syntax trees

Expressions in formal languages are represented by abstract syntax trees. An **abstract syntax tree** (or **AST** for short) is defined as either a leaf of one of the available leaf types, or a branch of one of the available branch types. All the available leaf and branch types must be specified as part of the definition of an AST. In other words, one must specify the data carried by leaves and branches, as well as the branching numbers.

To illustrate how ASTs are used, let us rewrite Example 3.2.2.4 via an AST. We view Example 3.2.2.4 as a small sub-language that deals with "safe integers" and supports the "safe arithmetic" operations `Sqrt`, `Add`, `Mul`, and `Div`. Example calculations in this sub-language are $\sqrt{16} * (1 + 2) = 12$; $20 + 1/0 =$ error; and $10 + \sqrt{-1} =$ error.

We can implement this sub-language in two stages. The first stage will create a data structure (an AST) that represents an unevaluated expression in the sub-language. The second stage will evaluate that AST to obtain either a number or an error message.

A straightforward way of defining a data structure for an AST is to use a disjunctive type whose parts describe all the possible operations of the sub-language. We will need one case class for each of `Sqrt`, `Add`, `Mul`, and `Div`. An additional operation, `Num`, will lift ordinary integers into "safe integers". So, we define the disjunctive type for "arithmetic sub-language expressions" as

```
sealed trait Arith
final case class Num(x: Int)                extends Arith
final case class Sqrt(x: Arith)             extends Arith
final case class Add(x: Arith, y: Arith)    extends Arith
final case class Mul(x: Arith, y: Arith)    extends Arith
final case class Div(x: Arith, y: Arith)    extends Arith
```

A value of type `Arith` is either a `Num(x)` for some integer `x`, or an `Add(x, y)` where `x` and `y` are previously defined `Arith` expressions, or another operation.

This type definition is similar to the binary tree type

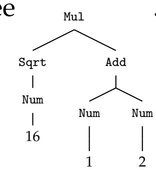
```
sealed trait Tree
final case class Leaf(x: Int)                 extends Tree
final case class Branch(x: Tree, y: Tree)     extends Tree
```

if we rename `Leaf` to `Num` and `Branch` to `Add`. However, the `Arith` type contains four different types of "branches", some with branching number 1 and others with branching number 2.

This example illustrates the structure of an AST: it is a tree of a general shape, where leaves and branches are chosen from a specified set of allowed possibilities. In this example, we have a single allowed type of leaf (`Num`) and four allowed types of branches (`Sqrt`, `Add`, `Mul`, and `Div`).

This completes the first stage of implementing the sub-language. We may now use the disjunctive type `Arith` to create expressions in the sub-language. For example, $\sqrt{16} * (1 + 2)$ is represented by

```
scala> val x: Arith = Mul(Sqrt(Num(16)), Add(Num(1), Num(2)))
x: Arith = Mul(Sqrt(Num(16)),Add(Num(1),Num(2)))
```

We can visualize `x` as the abstract syntax tree 

.

The expressions $20 + 1/0$ and $10 * \sqrt{-1}$ are represented by

```
scala> val y: Arith = Add(Num(20), Div(Num(1), Num(0)))
y: Arith = Add(Num(20),Div(Num(1),Num(0)))

scala> val z: Arith = Add(Num(10), Sqrt(Num(-1)))
z: Arith = Add(Num(10),Sqrt(Num(-1)))
```

As we see, the expressions `x`, `y`, and `z` *remain unevaluated*; each of them is a data structure that encodes a tree of operations of the sub-language. These operations will be evaluated at the second stage of implementing the sub-language.

To evaluate expressions in the "safe arithmetic", we can implement a function with type signature `run: Arith => Either[String, Int]`. That function plays the role of an **interpreter** or "**runner**" for programs written in the sub-language. The runner will destructure the expression tree and execute all the operations, taking care of possible errors.

To implement `run`, we need to define required arithmetic operations on the type `Either[String, Int]`. For instance, we need to be able to add or multiply values of that type. Instead of custom code from Example 3.2.2.4, we can use the standard `map` and `flatMap` methods defined on `Either`. For example, addition and multiplication of two "safe integers" is implemented as

```
def add(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 + r2) }
def mul(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.map(r2 => r1 * r2) }
```

while the "safe division" is

```
def div(x: Either[String, Int], y: Either[String, Int]):
    Either[String, Int] = x.flatMap { r1 => y.flatMap(r2 =>
  if (r2 == 0) Left(s"error: $r1 / $r2") else Right(r1 / r2) )
}
```

With this code, we can implement the runner as a recursive function,

```
def run: Arith => Either[String, Int] = {
  case Num(x)     => Right(x)
  case Sqrt(x)    => run(x).flatMap { r =>
    if (r < 0) Left(s"error: sqrt($r)") else Right(math.sqrt(r).toInt)
  }
  case Add(x, y)  => add(run(x), run(y))
  case Mul(x, y)  => mul(run(x), run(y))
  case Div(x, y)  => div(run(x), run(y))
}
```

Test it with the values `x`, `y`, `z` defined previously:

```
scala> run(x)
res0: Either[String, Int] = Right(12)

scala> run(y)
res1: Either[String, Int] = Left("error: 1 / 0")

scala> run(z)
res2: Either[String, Int] = Left("error: sqrt(-1)")
```