



Structure and Interpretation of Computer Programs – JavaScript Adaptation

Harold Abelson and Gerald Jay Sussman

with Julie Sussman

— authors

Martin Henz and Tobias Wrigstad

with Liu Hang, Feng Piaopiao, Jolyn Tan and Chan Ger Hean

— adapters to JavaScript

Contents

Foreword	5
Prefaces	9
Acknowledgments	14
1 Building Abstractions with Functions	17
1.1 The Elements of Programming	19
1.1.1 Expressions	20
1.1.2 Naming and the Environment	22
1.1.3 Evaluating Operator Combinations	23
1.1.4 Functions	25
1.1.5 The Substitution Model for Function Application	27
1.1.6 Conditional Expressions and Predicates	30
1.1.7 Example: Square Roots by Newton's Method	34
1.1.8 Functions as Black-Box Abstractions	38
1.2 Functions and the Processes They Generate	43
1.2.1 Linear Recursion and Iteration	44
1.2.2 Tree Recursion	49
1.2.3 Orders of Growth	54
1.2.4 Exponentiation	56
1.2.5 Greatest Common Divisors	59
1.2.6 Example: Testing for Primality	61
1.3 Formulating Abstractions with Higher-Order Functions	68
1.3.1 Functions as Arguments	69
1.3.2 Function Definition Expressions	74
1.3.3 Functions as General Methods	79
1.3.4 Functions as Returned Values	84

2 Building Abstractions with Data	91
2.1 Introduction to Data Abstraction	94
2.1.1 Example: Arithmetic Operations for Rational Numbers	95
2.1.2 Abstraction Barriers	99
2.1.3 What Is Meant by Data?	102
2.1.4 Extended Exercise: Interval Arithmetic	105
2.2 Hierarchical Data and the Closure Property	109
2.2.1 Representing Sequences	111
2.2.2 Hierarchical Structures	118
2.2.3 Sequences as Conventional Interfaces	125
2.2.4 Example: A Picture Language	140
2.3 Symbolic Data	154
2.3.1 Strings	154
2.3.2 Example: Symbolic Differentiation	157
2.3.3 Example: Representing Sets	163
2.3.4 Example: Huffman Encoding Trees	173
2.4 Multiple Representations for Abstract Data	182
2.4.1 Representations for Complex Numbers	184
2.4.2 Tagged data	188
2.4.3 Data-Directed Programming and Additivity	192
2.5 Systems with Generic Operations	201
2.5.1 Generic Arithmetic Operations	202
2.5.2 Combining Data of Different Types	207
2.5.3 Example: Symbolic Algebra	216
3 Modularity, Objects, and State	230
3.1 Assignment and Local State	231
3.1.1 Local State Variables	232
3.1.2 The Benefits of Introducing Assignment	238
3.1.3 The Costs of Introducing Assignment	242
3.2 The Environment Model of Evaluation	248
3.2.1 The Rules for Evaluation	250
3.2.2 Applying Simple Functions	253
3.2.3 Frames as the Repository of Local State	255
3.2.4 Internal Definitions	261

3.3	Modeling with Mutable Data	264
3.3.1	Mutable List Structure	265
3.3.2	Representing Queues	274
3.3.3	Representing Tables	279
3.3.4	A Simulator for Digital Circuits	286
3.3.5	Propagation of Constraints	298
3.4	Concurrency: Time Is of the Essence	310
3.4.1	The Nature of Time in Concurrent Systems	311
3.4.2	Mechanisms for Controlling Concurrency	316
3.5	Streams	330
3.5.1	Streams Are Delayed Lists	331
3.5.2	Infinite Streams	339
3.5.3	Exploiting the Stream Paradigm	347
3.5.4	Streams and Delayed Evaluation	358
3.5.5	Modularity of Functional Programs and Modularity of Objects	364
4	Metalinguistic Abstraction	370
4.1	The Metacircular Evaluator	372
4.1.1	The Core of the Evaluator	374
4.1.2	Representing Statements and Expressions	381
4.1.3	Evaluator Data Structures	386
4.1.4	Running the Evaluator as a Program	391
4.1.5	Data as Programs	395
4.1.6	Internal Declarations	398
4.1.7	Separating Syntactic Analysis from Execution	403
4.2	Lazy Evaluation	409
4.2.1	Normal Order and Applicative Order	409
4.2.2	An Interpreter with Lazy Evaluation	411
4.2.3	Streams as Lazy Lists	419
List Of Exercises		421
Solution To Exercises		431
References		488

Index	494
JavaScript Adaptation Making-of	514

Foreword

Educators, generals, dieticians, psychologists, and parents program. Armies, students, and some societies are programmed. An assault on large problems employs a succession of programs, most of which spring into existence en route. These programs are rife with issues that appear to be particular to the problem at hand. To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs—many of them. It doesn't matter much what the programs are about or what applications they serve. What does matter is how well they perform and how smoothly they fit with other programs in the creation of still greater programs. The programmer must seek both perfection of part and adequacy of collection. In this book the use of 'program' is focused on the creation, execution, and study of programs written in a dialect of Lisp for execution on a digital computer. Using Lisp we restrict or limit not what we may program, but only the notation for our program descriptions.

Our traffic with the subject matter of this book involves us with three foci of phenomena: the human mind, collections of computer programs, and the computer. Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from human experience and thought, are huge in number, intricate in detail, and at any time only partially understood. They are modeled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalizes until the model ultimately attains a metastable place within still another model with which we struggle. The source of the exhilaration associated with computer programming is the continual unfolding within the mind and on the computer of mechanisms expressed as programs and the explosion of perception they generate. If art interprets our dreams, the computer executes them in the guise of programs!

For all its power, the computer is a harsh taskmaster. Its programs must be correct, and what we wish to say must be said accurately in every detail. As in every other symbolic activity, we become convinced of program truth through argument. Lisp itself can be assigned a semantics (another model, by the way), and if a program's function can be specified, say, in the predicate calculus, the proof methods of logic can be used to make an acceptable correctness

argument. Unfortunately, as programs get large and complicated, as they almost always do, the adequacy, consistency, and correctness of the specifications themselves become open to doubt, so that complete formal arguments of correctness seldom accompany large programs. Since large programs grow from small ones, it is crucial that we develop an arsenal of standard program structures of whose correctness we have become sure—we call them idioms—and learn to combine them into larger structures using organizational techniques of proven value. These techniques are treated at length in this book, and understanding them is essential to participation in the Promethean enterprise called programming. More than anything else, the uncovering and mastery of powerful organizational techniques accelerates our ability to create large, significant programs. Conversely, since writing large programs is very taxing, we are stimulated to invent new methods of reducing the mass of function and detail to be fitted into large programs.

Unlike programs, computers must obey the laws of physics. If they wish to perform rapidly—a few nanoseconds per state change—they must transmit electrons only small distances (at most $1\frac{1}{2}$ feet). The heat generated by the huge number of devices so concentrated in space has to be removed. An exquisite engineering art has been developed balancing between multiplicity of function and density of devices. In any event, hardware always operates at a level more primitive than that at which we care to program. The processes that transform our Lisp programs to ‘machine’ programs are themselves abstract models which we program. Their study and creation give a great deal of insight into the organizational programs associated with programming arbitrary models. Of course the computer itself can be so modeled. Think of it: the behavior of the smallest physical switching element is modeled by quantum mechanics described by differential equations whose detailed behavior is captured by numerical approximations represented in computer programs executing on computers composed of ...!

It is not merely a matter of tactical convenience to separately identify the three foci. Even though, as they say, it’s all in the head, this logical separation induces an acceleration of symbolic traffic between these foci whose richness, vitality, and potential is exceeded in human experience only by the evolution of life itself. At best, relationships between the foci are metastable. The computers are never large enough or fast enough. Each breakthrough in hardware technology leads to more massive programming enterprises, new organizational principles, and an enrichment of abstract models. Every reader should ask himself periodically ‘Toward what end, toward what end?’—but do not ask it too often lest you pass up the fun of programming for the constipation of bittersweet philosophy.

Among the programs we write, some (but never enough) perform a precise mathematical function such as sorting or finding the maximum of a sequence of numbers, determining primality, or finding the square root. We call such programs algorithms, and a great deal is known of their optimal behavior, particularly with respect to the two important parameters of execution time and data storage requirements. A programmer should acquire good algorithms

and idioms. Even though some programs resist precise specifications, it is the responsibility of the programmer to estimate, and always to attempt to improve, their performance.

Lisp is a survivor, having been in use for about a quarter of a century. Among the active programming languages only Fortran has had a longer life. Both languages have supported the programming needs of important areas of application, Fortran for scientific and engineering computation and Lisp for artificial intelligence. These two areas continue to be important, and their programmers are so devoted to these two languages that Lisp and Fortran may well continue in active use for at least another quarter-century.

Lisp changes. The Scheme dialect used in this text has evolved from the original Lisp and differs from the latter in several important ways, including static scoping for variable binding and permitting functions to yield functions as values. In its semantic structure Scheme is as closely akin to Algol 60 as to early Lisps. Algol 60, never to be an active language again, lives on in the genes of Scheme and Pascal. It would be difficult to find two languages that are the communicating coin of two more different cultures than those gathered around these two languages. Pascal is for building pyramids—imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms—imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place. The organizing principles used are the same in both cases, except for one extraordinarily important difference: The discretionary exportable functionality entrusted to the individual Lisp programmer is more than an order of magnitude greater than that to be found within Pascal enterprises. Lisp programs inflate libraries with functions whose utility transcends the application that produced them. The list, Lisp's native data structure, is largely responsible for such growth of utility. The simple structure and natural applicability of lists are reflected in functions that are amazingly nonidiosyncratic. In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. As a result the pyramid must stand unchanged for a millennium; the organism must evolve or perish.

To illustrate this difference, compare the treatment of material and exercises within this book with that in any first-course text using Pascal. Do not labor under the illusion that this is a text digestible at MIT only, peculiar to the breed found there. It is precisely what a serious book on programming Lisp must be, no matter who the student is or where it is used.

Note that this is a text about programming, unlike most Lisp books, which are used as a preparation for work in artificial intelligence. After all, the critical programming concerns of software engineering and artificial intelligence tend to coalesce as the systems under investigation become larger. This explains why there is such growing interest in Lisp outside of artificial intelligence.

As one would expect from its goals, artificial intelligence research generates many significant programming problems. In other programming cultures this spate of problems spawns new languages. Indeed, in any very large programming task a useful organizing principle is to control and isolate traffic within the task modules via the invention of language. These languages tend to become less primitive as one approaches the boundaries of the system where we humans interact most often. As a result, such systems contain complex language-processing functions replicated many times. Lisp has such a simple syntax and semantics that parsing can be treated as an elementary task. Thus parsing technology plays almost no role in Lisp programs, and the construction of language processors is rarely an impediment to the rate of growth and change of large Lisp systems. Finally, it is this very simplicity of syntax and semantics that is responsible for the burden and freedom borne by all Lisp programmers. No Lisp program of any size beyond a few lines can be written without being saturated with discretionary functions. Invent and fit; have fits and reinvent! We toast the Lisp programmer who pens his thoughts within nests of parentheses.

— Alan J. Perlis, New Haven, Connecticut

Prefaces

Preface of JavaScript Adaptation

You are reading the textbook *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Sussman—with a twist. The textbook emphasizes the importance of abstraction for managing complexity, and introduces the reader to a host of concepts that lie at the heart of the field of computer science. Most of these ideas are independent of the programming language used to express them to employ actual computers for solving computational problems. They are programming-language independent. The twist then consists of replacing the programming language that is used in the examples. While the authors had used the programming language Scheme, this adaptation uses the language JavaScript.

More precisely, this adaptation uses five tiny, carefully designed, sublanguages of JavaScript. The languages are called *Source §1*, *Source §2*, *Source §3*, *Source §4* and *Source §5*, corresponding to the respective chapters 1, 2, 3, 4 and 5 of the textbook. The Source §1 language contains only constructs that are needed in the programs contained in chapter 1: constructs required to *build abstractions with functions*. Source §2 is a superset of Source §1; adding features required to build abstractions with data, on top of the features of Source §1. Similarly, Source §3, 4 and 5 extend the previous language features required to address the subject of the respective textbook chapter. All these languages are sub-languages of JavaScript; any Source program is also a JavaScript program. The reverse is not true. The JavaScript language has many features that are not covered in this textbook. Indeed, the Source languages are so small that they can be quite adequately described in a few pages of text. The online folder [source](#) contains the specifications of the Source languages, as reference for the reader.

This textbook is interactive. Most programs are links. Clicking on them takes the reader to a web-based programming environment called the [Source Academy](#). In the Source Academy, the reader can run the programs, modify them and experiment with them, without the need to install any software, and without any requirements on the computer that they use, as long as it comes with an internet browser.

The language Scheme has been designed as a sublanguage of Lisp with its use in education as

a central design objective. The language JavaScript, on the other hand, was not designed with the needs of learners in mind. This makes it difficult to use JavaScript in a course, even if one imposes constraints on the language features to be covered. The reason is that students with prior knowledge of the language are bound to make use of other features in their programs. Fellow students will legitimately ask the instructors about those features, and any answer will either frustrate the student or lead to a tangent that is most likely not conducive to the learning objectives. This problem is especially severe for JavaScript, which is not known for its systematic design. Our solution to this challenge is radical: The Source Academy *enforces* the use of the respective Source language when the student clicks on a program of a particular chapter. Programs that use constructs beyond that language are rejected by the Source Academy. This allows instructors of a SICP-based course to adopt JavaScript—one of the most widely used programming languages today—without getting bogged down in JavaScript’s plethora of idiosyncratic features.

The original textbook was introduced to the National University of Singapore by Jacob Katzenelson in 1997, as a more advanced alternative to the regular ‘Programming Methodology’ course offered to computer science students. The course, known as ‘CS1101S’ since 1998, switched to JavaScript in 2012, and became the required freshmen programming methodology course for Computer Science undergraduate majors in 2018.

— Martin Henz

Preface to the Second Edition

Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to always see it as a soap bubble?

— Alan J. Perlis

The material in this book has been the basis of MIT’s entry-level computer science subject since 1980. We had been teaching this material for four years when the first edition was published, and twelve more years have elapsed until the appearance of this second edition. We are pleased that our work has been widely adopted and incorporated into other texts. We have seen our students take the ideas and programs in this book and build them in as the core of new computer systems and languages. In literal realization of an ancient Talmudic pun, our students have become our builders. We are lucky to have such capable students and such accomplished builders.

In preparing this edition, we have incorporated hundreds of clarifications suggested by our own teaching experience and the comments of colleagues at MIT and elsewhere. We have re-

designed most of the major programming systems in the book, including the generic-arithmetic system, the interpreters, the register-machine simulator, and the compiler; and we have re-written all the program examples to ensure that any Scheme implementation conforming to the IEEE Scheme standard (IEEE 1990) will be able to run the code.

This edition emphasizes several new themes. The most important of these is the central role played by different approaches to dealing with time in computational models: objects with state, concurrent programming, functional programming, lazy evaluation, and nondeterministic programming. We have included new sections on concurrency and nondeterminism, and we have tried to integrate this theme throughout the book.

The first edition of the book closely followed the syllabus of our MIT one-semester subject. With all the new material in the second edition, it will not be possible to cover everything in a single semester, so the instructor will have to pick and choose. In our own teaching, we sometimes skip the section on logic programming , we have students use the register-machine simulator but we do not cover its implementation , and we give only a cursory overview of the compiler . Even so, this is still an intense course. Some instructors may wish to cover only the first three or four chapters, leaving the other material for subsequent courses.

The World-Wide-Web site [of MIT Press](#) provides support for users of this book. This includes programs from the book, sample programming assignments, supplementary materials, and downloadable implementations of the Scheme dialect of Lisp.

— Harold Abelson and Gerald Jay Sussman

Preface to the First Edition

A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists. Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.

— Marvin Minsky, ‘Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily-Formulated Ideas’

‘The Structure and Interpretation of Computer Programs’ is the entry-level subject in computer science at the Massachusetts Institute of Technology. It is required of all students at MIT who major in electrical engineering or in computer science, as one-fourth of the ‘common core

curriculum,’ which also includes two subjects on circuits and linear systems and a subject on the design of digital systems. We have been involved in the development of this subject since 1978, and we have taught this material in its present form since the fall of 1980 to between 600 and 700 students each year. Most of these students have had little or no prior formal training in computation, although many have played with computers a bit and a few have had extensive programming or hardware-design experience.

Our design of this introductory computer-science subject reflects two major concerns. First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems.

Our goal is that students who complete this subject should have a good feel for the elements of style and the aesthetics of programming. They should have command of the major techniques for controlling complexity in a large system. They should be capable of reading a 50-page-long program, if it is written in an exemplary style. They should know what not to read, and what they need not understand at any moment. They should feel secure about modifying a program, retaining the spirit and style of the original author.

These skills are by no means unique to computer programming. The techniques we teach and draw upon are common to all of engineering design. We control complexity by building abstractions that hide details when appropriate. We control complexity by establishing conventional interfaces that enable us to construct systems by combining standard, well-understood pieces in a ‘mix and match’ way. We control complexity by establishing new languages for describing a design, each of which emphasizes particular aspects of the design and deemphasizes others.

Underlying our approach to this subject is our conviction that ‘computer science’ is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology*—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of ‘what is.’ Computation provides a framework for dealing precisely with notions of ‘how to.’

In teaching our material we use a dialect of the programming language Lisp. We never formally teach the language, because we don’t have to. We just use it, and students pick it up in a few

days. This is one great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. All of the formal properties can be covered in an hour, like the rules of chess. After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues—figuring out what we want to compute, how we will decompose problems into manageable parts, and how we will work on the parts. Another advantage of Lisp is that it supports (but does not enforce) more of the large-scale strategies for modular decomposition of programs than any other language we know. We can make procedural and data abstractions, we can use higher-order functions to capture common patterns of usage, we can model local state using assignment and data mutation, we can link parts of a program with streams and delayed evaluation, and we can easily implement embedded languages. All of this is embedded in an interactive environment with excellent support for incremental program design, construction, testing, and debugging. We thank all the generations of Lisp wizards, starting with John McCarthy, who have fashioned a fine tool of unprecedented power and elegance.

Scheme, the dialect of Lisp that we use, is an attempt to bring together the power and elegance of Lisp and Algol. From Lisp we take the metalinguistic power that derives from the simple syntax, the uniform representation of programs as data objects, and the garbage-collected heap-allocated data. From Algol we take lexical scoping and block structure, which are gifts from the pioneers of programming-language design who were on the Algol committee. We wish to cite John Reynolds and Peter Landin for their insights into the relationship of Church’s lambda calculus to the structure of programming languages. We also recognize our debt to the mathematicians who scouted out this territory decades before computers appeared on the scene. These pioneers include Alonzo Church, Barkley Rosser, Stephen Kleene, and Haskell Curry.

— Harold Abelson and Gerald Jay Sussman

Acknowledgments

We would like to thank the many people who have helped us develop this book and this curriculum.

Our subject is a clear intellectual descendant of ‘6.231,’ a wonderful subject on programming linguistics and the lambda calculus taught at MIT in the late 1960s by Jack Wozencraft and Arthur Evans, Jr.

We owe a great debt to Robert Fano, who reorganized MIT’s introductory curriculum in electrical engineering and computer science to emphasize the principles of engineering design. He led us in starting out on this enterprise and wrote the first set of subject notes from which this book evolved.

Much of the style and aesthetics of programming that we try to teach were developed in conjunction with Guy Lewis Steele Jr., who collaborated with Gerald Jay Sussman in the initial development of the Scheme language. In addition, David Turner, Peter Henderson, Dan Friedman, David Wise, and Will Clinger have taught us many of the techniques of the functional programming community that appear in this book.

Joel Moses taught us about structuring large systems. His experience with the Macsyma system for symbolic computation provided the insight that one should avoid complexities of control and concentrate on organizing the data to reflect the real structure of the world being modeled.

Marvin Minsky and Seymour Papert formed many of our attitudes about programming and its place in our intellectual lives. To them we owe the understanding that computation provides a means of expression for exploring ideas that would otherwise be too complex to deal with precisely. They emphasize that a student’s ability to write and modify programs provides a powerful medium in which exploring becomes a natural activity.

We also strongly agree with Alan Perlis that programming is lots of fun and we had better be careful to support the joy of programming. Part of this joy derives from observing great masters at work. We are fortunate to have been apprentice programmers at the feet of Bill Gosper and Richard Greenblatt.

It is difficult to identify all the people who have contributed to the development of our curriculum. We thank all the lecturers, recitation instructors, and tutors who have worked with

us over the past fifteen years and put in many extra hours on our subject, especially Bill Siebert, Albert Meyer, Joe Stoy, Randy Davis, Louis Braida, Eric Grimson, Rod Brooks, Lynn Stein, and Peter Szolovits. We would like to specially acknowledge the outstanding teaching contributions of Franklyn Turbak, now at Wellesley; his work in undergraduate instruction set a standard that we can all aspire to. We are grateful to Jerry Saltzer and Jim Miller for helping us grapple with the mysteries of concurrency, and to Peter Szolovits and David McAllester for their contributions to the exposition of nondeterministic evaluation in chapter 4.

Many people have put in significant effort presenting this material at other universities. Some of the people we have worked closely with are Jacob Katzenelson at the Technion, Hardy Mayer at the University of California at Irvine, Joe Stoy at Oxford, Elisha Sacks at Purdue, and Jan Komorowski at the Norwegian University of Science and Technology. We are exceptionally proud of our colleagues who have received major teaching awards for their adaptations of this subject at other universities, including Kenneth Yip at Yale, Brian Harvey at the University of California at Berkeley, and Dan Huttenlocher at Cornell.

Al Moyée arranged for us to teach this material to engineers at Hewlett-Packard, and for the production of videotapes of these lectures. We would like to thank the talented instructors—in particular Jim Miller, Bill Siebert, and Mike Eisenberg—who have designed continuing education courses incorporating these tapes and taught them at universities and industry all over the world.

Many educators in other countries have put in significant work translating the first edition. Michel Briand, Pierre Chamard, and André Pic produced a French edition; Susanne Daniels-Herold produced a German edition; and Fumio Motoyoshi produced a Japanese edition. We do not know who produced the Chinese edition, but we consider it an honor to have been selected as the subject of an ‘unauthorized’ translation.

It is hard to enumerate all the people who have made technical contributions to the development of the Scheme systems we use for instructional purposes. In addition to Guy Steele, principal wizards have included Chris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas, and Stephen Adams. Others who have put in significant time are Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larrabee, George Cartette, Soma Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Todd Cass, Patrick O’Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtemanche, Henry M. Wu, Andrew Berlin, and Ruth Shyu.

Beyond the MIT implementation, we would like to thank the many people who worked on the IEEE Scheme standard, including William Clinger and Jonathan Rees, who edited the R4RS, and Chris Haynes, David Bartley, Chris Hanson, and Jim Miller, who prepared the IEEE standard.

Dan Friedman has been a long-time leader of the Scheme community. The community’s broader work goes beyond issues of language design to encompass significant educational

innovations, such as the high-school curriculum based on EdScheme by Schemer's Inc., and the wonderful books by Mike Eisenberg and by Brian Harvey and Matthew Wright.

We appreciate the work of those who contributed to making this a real book, especially Terry Ehling, Larry Cohen, and Paul Bethge at the MIT Press. Ella Mazel found the wonderful cover image. For the second edition we are particularly grateful to Bernard and Ella Mazel for help with the book design, and to David Jones, TeX wizard extraordinaire. We also are indebted to those readers who made penetrating comments on the new draft: Jacob Katzenelson, Hardy Mayer, Jim Miller, and especially Brian Harvey, who did unto this book as Julie did unto his book *Simply Scheme*.

Finally, we would like to acknowledge the support of the organizations that have encouraged this work over the years, including support from Hewlett-Packard, made possible by Ira Goldstein and Joel Birnbaum, and support from DARPA, made possible by Bob Kahn.

— Harold Abelson and Gerald Jay Sussman

Chapter 1

Building Abstractions with Functions

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.

— John Locke, *An Essay Concerning Human Understanding* (1690)

We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called *data*. The evolution of a process is directed by a pattern of rules called a *program*. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.

A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring. Even small errors (usually called *bugs* or *glitches*) in programs can have complex and unanticipated consequences.

Fortunately, learning to program is considerably less dangerous than learning sorcery, because the spirits we deal with are conveniently contained in a secure way. Real-world programming, however, requires care, expertise, and wisdom. A small bug in a computer-aided design program, for example, can lead to the catastrophic collapse of an airplane or a dam or the self-destruction of an industrial robot.

Master software engineers have the ability to organize programs so that they can be reasonably sure that the resulting processes will perform the tasks intended. They can visualize the behavior of their systems in advance. They know how to structure programs so that unanticipated problems do not lead to catastrophic consequences, and when problems do arise, they can *debug* their programs. Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be constructed, replaced, and debugged separately.

Programming in JavaScript

We need an appropriate language for describing processes, and we will use for this purpose the programming language JavaScript. Just as our everyday thoughts are usually expressed in our natural language (such as English, French, or Japanese), and descriptions of quantitative phenomena are expressed with mathematical notations, our process descriptions will be expressed in JavaScript. JavaScript was developed in the early 1990s as a programming language for controlling the behavior of World Wide Web browsers through scripts that are embedded in web pages. The language was conceived by Brendan Eich, originally under the name *Mocha*, which was later renamed to *LiveScript*, and finally to JavaScript. The name ‘JavaScript’ is a trademark of Oracle Corporation.

Despite its inception as a language for controlling browsers, JavaScript is a general-purpose programming language. A JavaScript *interpreter* is a machine that carries out processes described in the JavaScript language. The first JavaScript interpreter was implemented by Eich at Netscape Communications Corporation, for the Netscape Navigator web browser. The main features of JavaScript are inherited from the Scheme and Self programming languages. Scheme is a dialect of Lisp, and was used as programming language for the original version of this book. From Scheme, JavaScript inherited its most fundamental design principles such as statically-scoped first-class functions and dynamic typing, and as a result, it was fairly straightforward to translate the programs in this book from Scheme to JavaScript.

JavaScript bears only superficial resemblance to the language Java, after which it was (eventually) named; both Java and JavaScript use the block structure of the language C. In contrast with Java and C, which usually employ compilation to lower-level languages, JavaScript programs were initially *interpreted* by web browsers. After Netscape Navigator, other web browsers provided interpreters for the language, including Microsoft’s Internet Explorer, whose JavaS-

cript version is called *JScript*. The popularity of JavaScript for controlling web browsers gave rise to a standardization effort, culminating in *ECMAScript*. The first edition of the ECMAScript standard was led by Guy Lewis Steele Jr. and completed in June 1997 (Ecma 1997). The sixth edition, which is used in this book, was led by Allen Wirfs-Brock and adopted by the General Assembly of ECMA in June 2015.

The practice of embedding JavaScript programs in web pages encouraged the developers of web browsers to implement JavaScript interpreters. As these programs became more complex, the interpreters became more efficient in executing them, eventually using sophisticated implementation techniques such as Just-In-Time (JIT) compilation. The majority of JavaScript programs (as of 2019) is embedded in web pages and interpreted by browsers, but JavaScript is also used for scripting dashboard widgets in Apple computers running the OS X operating system, for controlling software systems such as Adobe Reader and devices such as universal remote panels, and in server software, using Node.js.

However, it is the ability of browsers to execute JavaScript programs that makes it an ideal language for an online version of a programming textbook. Executing programs by clicking on things on a web page comes naturally in JavaScript—after all that is what JavaScript was designed for! More fundamentally, JavaScript possesses features that make it an excellent medium for studying important programming constructs and data structures and for relating them to the linguistic features that support them. JavaScript’s statically-scoped first-class functions provide direct and concise access to abstraction mechanisms, and dynamic typing removes the need for declaring the types of the data being manipulated by the program. Above and beyond these considerations, programming in JavaScript is great fun.

1.1 The Elements of Programming

A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes. Thus, when we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- **primitive expressions**, which represent the simplest entities the language is concerned with,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

In programming, we deal with two kinds of elements: functions and data. (Later we will discover that they are really not so distinct.) Informally, data is ‘stuff’ that we want to manipulate, and functions are descriptions of the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive functions and should have methods for combining and abstracting functions and data.

In this chapter we will deal only with simple numerical data so that we can focus on the rules for building functions.¹ In later chapters we will see that these same rules allow us to build functions to manipulate compound data as well.

1.1.1 Expressions

One easy way to get started at programming in JavaScript is to interact with the JavaScript interpreter that is built into the browser you are using to view this page. We have set up the statements shown with a dark background such that you can click on them. The mouse click on JavaScript statements is programmed in such a way that a JavaScript interpreter is displayed, which can *evaluate* the statement and display the resulting value. By the way, the program that makes the mouse click on a JavaScript statement display the interpreter is itself written in JavaScript; we call it the *script* for the mouse click.

One kind of statement is an A simple kind of an expression is a number. (More precisely, the expression consists of the numerals that represent the number in base 10.) If you ask our script to display the interpreter for the expression statement

486;

by clicking it, it will respond by creating a separate browser tab where the statement is shown, with the option to evaluate the statement. Click on the primitive expression statement, and see what happens!

Expressions representing numbers may be combined with operators (such as + or *) to form a compound expression that represents the application of a corresponding primitive function to those numbers. For example, evaluate any of the following expression statements² by clicking

¹The characterization of numbers as ‘simple data’ is a barefaced bluff. In fact, the treatment of numbers is one of the trickiest and most confusing aspects of any programming language. Some typical issues involved are these: How large a number can we represent? How many decimal places of accuracy can we represent? Above and beyond these questions, of course, lies a collection of issues concerning roundoff and truncation errors—the entire science of numerical analysis. Since our focus in this book is on large-scale program design rather than on numerical techniques, we are going to ignore these problems. The numerical examples in this chapter will exhibit the usual roundoff behavior that one observes when using arithmetic operations that preserve a limited number of decimal places of accuracy in noninteger operations.

²Note that the semicolon indicates to the JavaScript interpreter that the expression should be taken as a statement, and thus as a complete program. However, JavaScript systems are not strict about these semicolons; they can often be left out. In this book, we will never leave out these optional semicolons, and point out which statements come with semicolons and which ones don’t.

on it:

```
137 + 349;  
1000 - 334;  
5 * 99;  
10 / 5;  
2.7 + 10;
```

Expressions such as these, which contain other expressions as components, are called *combinations*. Combinations that are formed by an *operator* symbol in the middle, and *operand* expressions to the left and right of it, are called *operator combinations*. The value of an operator combination is obtained by applying the function specified by the operator to the *arguments* that are the values of the operands.

The convention of placing the operator between the operands is known as *Infix notation*. It follows the mathematical notation that the reader is most likely familiar with from school and everyday life. As in mathematics, operator combinations can be *nested*, that is, they can take arguments that themselves are operator combinations:

```
(3 * 5) + (10 - 6);
```

As usual, parentheses are used to group operator combinations in order to avoid ambiguities. JavaScript also follows the usual conventions when parentheses are omitted; multiplication and division bind stronger than addition and subtraction. For example,

```
3 * 5 + 10 / 2;
```

stands for

```
(3 * 5) + (10 / 2);
```

We say that *** and */* have *higher precedence* than *+* and *-*. Sequences of additions and subtractions are read from left to right, as are sequences of multiplications and divisions. Thus,

```
3 / 5 * 2 - 4 + 3;
```

stands for

```
((3 / 5) * 2) - 4) + 3;
```

We say that the operators *+*, *-*, *** and */* are *left-associative*.

It is we humans who might get confused by still relatively simple expressions such as

```
3 * 2 * (4 + (3 - 5)) + 10 * (27 / 6);
```

which the interpreter would readily evaluate to be 57. We can help ourselves by writing such an expression in the form

```
3 * 2 * (4 + (3 - 5))
+
10 * (27 / 6);
```

to visually separate the major components of the expression.

The interpreter always operates in the same basic cycle: It reads a statement from the browser, evaluates the statement, and prints the result. This mode of operation is often expressed by saying that the interpreter runs in a *read-eval-print loop*. Observe in particular that it is not necessary to explicitly instruct the interpreter to print the value of the statement.

1.1.2 Naming and the Environment

A critical aspect of a programming language is the means it provides for using names to refer to computational objects, and our first such means are *constants*. We say that the name identifies a constant whose *value* is the object.

In JavaScript, we name constants using *constant declarations*. Typing

```
const size = 2;
```

causes the interpreter to associate the value 2 with the name `size`. The purpose of the constant declaration is to create this association, and not to compute a particular value as for expression statements. The JavaScript specification demands that the special value `undefined` shall be the result of evaluating such constant declarations.

Once the name `size` has been associated with the number 2, we can refer to the value 2 by name as in

```
size;
```

or

```
5 * size;
```

Here are further examples of the use of `const`:

```
const pi = 3.14159;
const radius = 10;
pi * radius * radius;
const circumference = 2 * pi * radius;
circumference;
```

Constant declaration is our language's simplest means of abstraction, for it allow us to use simple names to refer to the results of compound operations, such as the `circumference`

computed above. In general, computational objects may have very complex structures, and it would be extremely inconvenient to have to remember and repeat their details each time we want to use them. Indeed, complex programs are constructed by building, step by step, computational objects of increasing complexity.

It should be clear that the possibility of associating values with names and later retrieving them means that the interpreter must maintain some sort of memory that keeps track of the name-object pairs. This memory is called the *environment* (more precisely the *global environment*, since we will see later that a computation may involve a number of different environments).³

1.1.3 Evaluating Operator Combinations

One of our goals in this chapter is to isolate issues about process descriptions. As a case in point, let us consider that, in evaluating operator combinations, the interpreter proceeds as follows.

- To evaluate an operator combination, do the following:
 - Evaluate the operand expressions of the combination.
 - Apply the function that is denoted by the operator to the arguments that are the values of the operands.

Even this simple rule illustrates some important points about processes in general. First, observe that the first step dictates that in order to accomplish the evaluation process for an operator expression we must first perform the evaluation process on each operand of the operator combination. Thus, the evaluation rule is *recursive* in nature; that is, it includes, as one of its steps, the need to invoke the rule itself.

Notice how succinctly the idea of recursion can be used to express what, in the case of a deeply nested combination, would otherwise be viewed as a rather complicated process. For example, evaluating

```
(2 + 4 * 6) * (3 + 12);
```

requires that the evaluation rule be applied to four different combinations. We can obtain a picture of this process by representing the combination in the form of a tree, as shown in Figure 1.1. Each combination is represented by a node with branches corresponding to the operator and the operands of the operator combination stemming from it. The terminal nodes (that is, nodes with no branches stemming from them) represent either operators or numbers. Viewing evaluation in terms of the tree, we can imagine that the values of the operands percolate

³Chapter 3 will show that this notion of environment is crucial, both for understanding how the interpreter works and for implementing interpreters.

upward, starting from the terminal nodes and then combining at higher and higher levels. In general, we shall see that recursion is a very powerful technique for dealing with hierarchical, treelike objects. In fact, the ‘percolate values upward’ form of the evaluation rule is an example of a general kind of process known as *tree accumulation*.

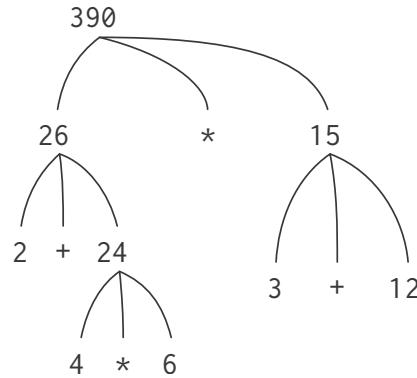


Figure 1.1: Tree representation, showing the value of each subexpression.

Next, observe that the repeated application of the first step brings us to the point where we need to evaluate, not operator combinations, but primitive expressions such as numerals or names. We take care of the primitive cases by stipulating that

Notice the role of the environment in determining the meaning of the names in expressions. In JavaScript, it is meaningless to speak of the value of an expression such as `x + 1` without specifying any information about the environment that would provide a meaning for the name `x`. As we shall see in chapter 3, the general notion of the environment as providing a context in which evaluation takes place will play an important role in our understanding of program execution.

Notice that the evaluation rule given above does not handle constant declarations. For instance, evaluating `const x = 3;` does not apply the `=` operator to two arguments, one of which is the value of the name `x` and the other of which is 3, since the purpose of the constant declaration is precisely to associate `x` with a value. (That is, the part `x = 3` in the constant declaration `const x = 3;` is not an operator combination.)

The string ‘`const`’ in the constant declaration is rendered in bold letters to indicate that it is a *keyword* in JavaScript. Keywords are reserved words that carry a particular meaning, and thus cannot be used as names. A keyword or a combination of keywords instructs the JavaScript interpreter to treat the respective statement in a special way. Each such syntactic form has its own evaluation rule. The various kinds of statements (each with its associated evaluation rule) constitute the syntax of the programming language.

1.1.4 Functions

We have identified in JavaScript some of the elements that must appear in any powerful programming language:

- Numbers and arithmetic operations are primitive data and functions.
- Nesting of combinations provides a means of combining operations.
- Constant declarations that associate names with values provide a limited means of abstraction.

Now we will learn about *function declarations*, a much more powerful abstraction technique by which a compound operation can be given a name and then referred to as a unit.

We begin by examining how to express the idea of ‘squaring.’ We might say, ‘To square something, take it times itself.’ This is expressed in our language as

```
function square(x) {
    return x * x;
}
```

We can understand this in the following way:

```
function square(      x      ) { return x      *      x; }
//   ^          ^          ^
// To      square something,   take      it times itself.
```

We have here a *compound function*, which has been given the name `square`. The function represents the operation of multiplying something by itself. The thing to be multiplied is given a local name, `x`, which plays the same role that a pronoun plays in natural language. Evaluating the declaration creates this compound function and associates it with the name `square`.⁴

Our simplest form of a function declaration is

```
function name( parameters ) { return expression; }
```

The *name* is a symbol to be associated with the function in the environment.⁵ The *parameters* are the names used within the body of the function to refer to the corresponding arguments of the function. The *expression* after the keyword `return` is the *return expression* that will yield the value of the function application when the parameters are replaced by the arguments to

⁴Observe that there are two different operations being combined here: we are creating the function, and we are giving it the name `square`. It is possible, indeed important, to be able to separate these two notions—to create functions without naming them, and to give names to functions that have already been created. We will see how to do this in section 1.3.2.

⁵Throughout this book, we will describe the general syntax of programs by using italic symbols—e.g., *name*—to denote the ‘slots’ in the expression to be filled in when such an expression is actually used.

which the function is applied.⁶ The *parameters* are grouped within parentheses and separated by commas, just as they would be in an actual call to the function being declared.

Having declared the `square` function, we can now use it in a *function application expression*, which we turn into a statement using a semicolon:

```
square(21);
```

The name `square` is the *function expression* of the application, and `21` is the *argument expression*.

```
square(2 + 5);
```

Here, the argument expression is itself a compound expression, the operator expression `2 + 5`.

```
square(square(3));
```

Of course application expressions can also serve as argument expressions.

We can also use `square` as a building block in declaring other functions. For example, $x^2 + y^2$ can be expressed as

```
square(x) + square(y);
```

We can easily declare a function `sum_of_squares` that, given any two numbers as arguments, produces the sum of their squares:

```
function sum_of_squares(x, y) {
    return square(x) + square(y);
}
```

Now we can use `sum_of_squares` as a building block in constructing further functions:

```
function f(a) {
    return sum_of_squares(a + 1, a * 2);
}
```

The application of functions such as `sum_of_squares(3, 4)` is—after operator combination—the second kind of combination of expressions into larger expressions that we encounter. In addition to compound functions, JavaScript provides a number of *primitive functions* that are built into the interpreter. An example is the function `math_log` that computes the natural logarithm of its argument.⁷ Evaluating the application expression `math_log(1)` results in the number `0`. Primitive functions are used in exactly the same way as compound functions. Indeed, one could not tell by looking at the definition of `sum_of_squares` given above whether `square` was built into the interpreter, like `math_log`, or defined as a compound function.

⁶We shall see in the next section that the body of the function can be a sequence of statements. In this case, the interpreter evaluates each statement in the sequence in turn until a `return` statement determines the value of the function application.

⁷The Source language used in this adaptation introduces names `math_*` for all functions and constants in JavaScript's [Math library](#).

1.1.5 The Substitution Model for Function Application

To evaluate an application combination, the interpreter follows a similar process as for operator combinations, which we described in section 1.1.3. That is, the interpreter evaluates the elements of the combination and applies the function (which is the value of the function expression) to the arguments (which are the values of the argument expressions of the application combination).

In more detail, the interpreter proceeds as follows when evaluating application combinations:

- To evaluate an application combination of the form

```
function-expression ( argument-expressions )
```

do the following:

- Evaluate the function expression of the application combination, resulting in the function to be applied.
- Evaluate the argument expressions of the combination.
- Apply the function to the arguments:
 - If the function is primitive, we simply apply the corresponding mathematical function to the arguments.
 - If the function is compound, we evaluate the return expression of the function with each parameter replaced by the corresponding argument.

To illustrate this process, let's evaluate the application combination

```
f(5);
```

Evaluating the name `f` results in the function declared in section 1.1.4. Evaluating the argument expression `5` yields the argument value `5`. Now, we retrieve the return expression of `f`:

```
sum_of_squares(a + 1, a * 2)
```

in which we replace the parameter `a` by the argument `5`:

```
sum_of_squares(5 + 1, 5 * 2)
```

Thus the problem reduces to the evaluation of an application combination with two arguments and a function expression `sum_of_squares`. Evaluating this combination involves three subproblems. We must evaluate the function expression to get the function to be applied, and we must evaluate the argument expressions to get the arguments. Now `5 + 1` produces `6` and `5 * 2` produces `10`, so we must apply the `sum_of_squares` function to `6` and `10`. These values are substituted for the parameters `x` and `y` in the return expression of `sum_of_squares`, reducing the expression to

```
square(6) + square(10)
```

If we use the declaration of `square` once, this reduces to

```
(6 * 6) + square(10)
```

which reduces by multiplication to

```
36 + square(10)
```

A second application of `square` yields

```
36 + (10 * 10)
```

which reduces by multiplication to

```
36 + 100
```

and finally by addition to

```
136
```

The process we have just described is called the *substitution model* for function application. It can be taken as a model that determines the ‘meaning’ of function application, insofar as the functions in this chapter are concerned. However, there are two points that should be stressed:

- The purpose of the substitution is to help us think about function application, not to provide a description of how the interpreter really works. Typical interpreters do not evaluate function applications by manipulating the text of a function to substitute values for the parameters. In practice, the ‘substitution’ is accomplished by using a local environment for the parameters. We will discuss this more fully in chapters 3 and 4 when we examine the implementation of an interpreter in detail.
- Over the course of this book, we will present a sequence of increasingly elaborate models of how interpreters work, culminating with a complete implementation of an interpreter and compiler in chapter 5. The substitution model is only the first of these models—a way to get started thinking formally about the evaluation process. In general, when modeling phenomena in science and engineering, we begin with simplified, incomplete models. As we examine things in greater detail, these simple models become inadequate and must be replaced by more refined models. The substitution model is no exception. In particular, when we address in chapter 3 the use of functions with ‘mutable data,’ we will see that the substitution model breaks down and must be replaced by a more complicated model of function application.⁸

⁸Despite the simplicity of the substitution idea, it turns out to be surprisingly complicated to give a rigorous mathematical definition of the substitution process. The problem arises from the possibility of confusion between the names used for the parameters of a function and the (possibly identical) names used in the expressions to which the function may be applied. Indeed, there is a long history of erroneous definitions of *substitution* in the literature of logic and programming semantics. See Stoy 1977 for a careful discussion of substitution.

Applicative order versus normal order

According to the description of evaluation given above, the interpreter first evaluates the function and argument expressions and then applies the resulting function to the resulting arguments. This is not the only way to perform evaluation. An alternative evaluation model would not evaluate the operands until their values were needed. Instead it would first substitute argument expressions for parameters until it obtained an expression involving only operators, and would then perform the evaluation. If we used this method, the evaluation of

$f(5)$

would proceed according to the sequence of expansions

```
sum_of_squares(5 + 1, 5 * 2)
square(5 + 1) + square(5 * 2)
(5 + 1) * (5 + 1) + square(5 * 2)
(5 + 1) * (5 + 1) + (5 * 2) * (5 * 2)
```

followed by the reductions

```
6 * (5 + 1) + (5 * 2) * (5 * 2)
6 * 6 + (5 * 2) * (5 * 2)
36 + (5 * 2) * (5 * 2)
36 + 10 * (5 * 2)
36 + 10 * 10
36 + 100
136
```

This gives the same answer as our previous evaluation model, but the process is different. In particular, the evaluations of $5 + 1$ and $5 * 2$ are each performed twice here, corresponding to the reduction of the expression

$x * x$

with x replaced respectively by $5 + 1$ and $5 * 2$.

This alternative ‘fully expand and then reduce’ evaluation method is known as *normal-order evaluation*, in contrast to the ‘evaluate the arguments and then apply’ method that the interpreter actually uses, which is called *applicative-order evaluation*. It can be shown that, for function applications that can be modeled using substitution (including all the functions in the first two chapters of this book) and that yield legitimate values, normal-order and applicative-order evaluation produce the same value. (See exercise 1.5 for an instance of an ‘illegitimate’ value where normal-order and applicative-order evaluation do not give the same result.)

JavaScript uses applicative-order evaluation, partly because of the additional efficiency obtained from avoiding multiple evaluations of expressions such as those illustrated with above and, more significantly, because normal-order evaluation becomes much more complicated to

deal with when we leave the realm of procedures that can be modeled by substitution. On the other hand, normal-order evaluation can be an extremely valuable tool, and we will investigate some of its implications in chapters 3 and 4.⁹

1.1.6 Conditional Expressions and Predicates

The expressive power of the class of functions that we can declare at this point is very limited, because we have no way to make tests and to perform different operations depending on the result of a test. For instance, we cannot declare a function that computes the absolute value of a number by testing whether the number is negative or not, and taking different actions in each case according to the rule

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

This construct is a *case analysis* and can be expressed in JavaScript using a *conditional expression* as follows:

```
function abs(x) {
    return x >= 0 ? x : -x;
}
```

The general form of a conditional expression is

```
predicate ? consequent-expression : alternative-expression
```

Conditional expressions begin with a *predicate*—that is, an expression whose value is interpreted as either *true* or *false*, two distinguished *boolean* values in JavaScript.¹⁰ Note that the primitive boolean expressions `true` and `false` trivially evaluate to the boolean values *true* and *false*, respectively. The *predicate* is followed by a question mark, the *consequent-expression*, a colon, and finally the *alternative-expression*.

To evaluate a conditional expression, the interpreter starts by evaluating the *predicate* part of the expression. If the *predicate* evaluates to *true*, the interpreter evaluates *consequent-expression*. Otherwise it evaluates *alternative-expression*.

The word *predicate* is used for functions that return *true* or *false*, as well as for expressions that evaluate to *true* or *false*. The absolute-value function `abs` makes use of the primitive predicate `>=`. This predicate takes two numbers as arguments and tests whether the first number is greater than or equal to the second number, returning *true* or *false* accordingly.

⁹In chapter 3 we will introduce *stream processing*, which is a way of handling apparently ‘infinite’ data structures by incorporating a limited form of normal-order evaluation. In section 4.2 we will modify the JavaScript interpreter to produce a normal-order variant of JavaScript.

¹⁰In JavaScript, other values are automatically converted into *true* and *false* according to *conversion rules*, but we choose not to make use of these conversion rules in this book.

JavaScript provides a number of primitive predicates that work similar to `>=`, including `>`, `<`, `<=`, and `==`. In addition to these primitive predicates, there are logical composition operations, which enable us to construct compound predicates. The three most frequently used are these:

- `expression1 && expression2` The interpreter evaluates `expression1`. If it evaluates to `false`, the value of the whole expression is `false`, and `expression2` is not evaluated. If `expression1` evaluates to `true`, the value of the whole expression is the value of `expression2`.
- `expression1 || expression2` The interpreter evaluates `expression1`. If it evaluates to `true`, the value of the whole expression is `true`, and `expression2` is not evaluated. If `expression1` evaluates to `false`, the value of the whole expression is the value of `expression2`.
- `! expression` The value of the expression is `true` when `expression` evaluates to `false`, and `false` otherwise.

Notice that `&&` and `||` are not evaluated like arithmetic operators such as `+`, because their right-hand expression is not always evaluated. The operator `!`, on the other hand, follows the evaluation rule of section 1.1.3. It is a *unary* operator, which means that it takes only one argument, whereas the arithmetic operators encountered so far are *binary*, taking two arguments. The operator `!` precedes its argument; we call it a *prefix operator*. Another prefix operator is the unary ‘minus’ operator, an example of which is the expression `-x` of the function `abs` in the beginning of this section.

As an example of how these predicates are used, the condition that a number `x` be in the range $5 < x < 10$ may be expressed as

```
x > 5 && x < 10
```

Note that the binary operator `&&` has lower precedence than the comparison operators `>` and `<`.

```
function not_equal(x, y) {
    return x > y || x < y;
}
```

or alternatively as

```
function not_equal(x, y) {
    return !(x >= y && x <= y);
}
```

Note that the operator `!==` when applied to two numbers, behaves the same as `not_equal`.

Exercise 1.1

Below is a sequence of statements. Before you click on a statement, predict what the result of its evaluation will be.

```

10;
5 + 3 + 4;
9 - 1;
6 / 2;
2 * 4 + (4 - 6);
const a = 3;
const b = a + 1;
a + b + a * b;
a === b;
b > a && b < a * b
? b : a;
a === 4 ? 6 : b === 4 ? 6 + 7 + a : 25;
2 + (b > a ? b : a);

(a > b
? a
: a < b
? b
: -1)
*
(a + 1);

```

Note that the statement

```
a === 4 ? 6 : b === 4 ? 6 + 7 + a : 25;
```

consists of two conditional expressions, where the second one forms the alternative of the first one. If you want to make that clear, you can indent the lines like this:

```

a === 4
? 6
: b === 4
? 6 + 7 + a
: 25;

```

Exercise 1.2

Translate the following expression into JavaScript

$$\frac{5 + 4 + (2 - (3 - (6 + \frac{4}{5}))))}{3(6 - 2)(2 - 7)}$$

[Solution](#)

Exercise 1.3

Declare a function that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

[Solution](#)

Exercise 1.4

Observe that our model of evaluation allows for application combinations whose function expressions are compound expressions. Use this observation to describe the behavior of the following function:

```
function plus(a, b) { return a + b; }
function minus(a, b) { return a - b; }
function a_plus_abs_b(a, b) {
    return (b >= 0 ? plus : minus)(a, b);
}
```

Note that in the conditional expression, we cannot directly use the operators + and - instead of the names plus and minus because in infix notation, only operator symbols are allowed in the middle, not compound expressions.¹¹

[Solution](#)

Exercise 1.5

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He declares the following two functions :

¹¹For an expression of the form $a (b > 0 ? + : -) b$ the JavaScript interpreter would not know the precedence of the operator between a and b, and therefore such expressions are not allowed.

```

function p() {
    return p();
}

function test(x, y) {
    return x === 0 ? 0 : y;
}

```

Then he evaluates the statement

```
test(0, p());
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for conditional expressions is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

[Solution](#)

1.1.7 Example: Square Roots by Newton's Method

Functions, as introduced above, are much like ordinary mathematical functions. They specify a value that is determined by one or more parameters. But there is an important difference between mathematical functions and computer functions. Computer functions must be effective.

As a case in point, consider the problem of computing square roots. We can define the square-root function as

$$\sqrt{x} = \text{the } y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

This describes a perfectly legitimate mathematical function. We could use it to recognize whether one number is the square root of another, or to derive facts about square roots in general. On the other hand, the definition does not describe a computer function. Indeed, it tells us almost nothing about how to actually find the square root of a given number. It will not help matters to rephrase this definition in pseudo-JavaScript:

```

function sqrt(x) {
    return the y with y >= 0 &&
        square(y) === x;
}

```

This only begs the question.

The contrast between mathematical function and computer function is a reflection of the general distinction between describing properties of things and describing how to do things,

or, as it is sometimes referred to, the distinction between declarative knowledge and imperative knowledge. In mathematics we are usually concerned with declarative (what is) descriptions, whereas in computer science we are usually concerned with imperative (how to) descriptions.¹²

How does one compute square roots? The most common way is to use Newton's method of successive approximations, which says that whenever we have a guess y for the value of the square root of a number x , we can perform a simple manipulation to get a better guess (one closer to the actual square root) by averaging y with x/y .¹³ For example, we can compute the square root of 2 as follows. Suppose our initial guess is 1:

Guess	Quotient	Average
1	$\frac{2}{1} = 2$	$\frac{(2+1)}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333$	$\frac{(1.3333+1.5)}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{(1.4167+1.4118)}{2} = 1.4142$
1.4142

Continuing this process, we obtain better and better approximations to the square root.

Now let's formalize the process in terms of functions. We start with a value for the radicand (the number whose square root we are trying to compute) and a value for the guess. If the guess is good enough for our purposes, we are done; if not, we must repeat the process with an improved guess. We write this basic strategy as a function:

¹²Declarative and imperative descriptions are intimately related, as indeed are mathematics and computer science. For instance, to say that the answer produced by a program is 'correct' is to make a declarative statement about the program. There is a large amount of research aimed at establishing techniques for proving that programs are correct, and much of the technical difficulty of this subject has to do with negotiating the transition between imperative statements (from which programs are constructed) and declarative statements (which can be used to deduce things). In a related vein, an important current area in programming-language design is the exploration of so-called very high-level languages, in which one actually programs in terms of declarative statements. The idea is to make interpreters sophisticated enough so that, given 'what is' knowledge specified by the programmer, they can generate 'how to' knowledge automatically. This cannot be done in general, but there are important areas where progress has been made. We shall revisit this idea in chapter 4.

¹³This square-root algorithm is actually a special case of Newton's method, which is a general technique for finding roots of equations. The square-root algorithm itself was developed by Heron of Alexandria in the first century A.D. We will see how to express the general Newton's method as a JavaScript function in section 1.3.4.

```
function sqrt_iter(guess, x) {
    return good_enough(guess, x)
        ? guess
        : sqrt_iter(improve(guess, x), x);
}
```

A guess is improved by averaging it with the quotient of the radicand and the old guess:

```
function improve(guess, x) {
    return average(guess, x / guess);
}
```

where

```
function average(x,y) {
    return (x + y) / 2;
}
```

We also have to say what we mean by ‘good enough.’ The following will do for illustration, but it is not really a very good test. (See exercise 1.7.)

```
function good_enough(guess, x) {
    return abs(square(guess) - x) < 0.001;
}

function sqrt(x) {
    return sqrt_iter(1, x);
}
```

If we type these declarations to the interpreter, we can use `sqrt` just as we can use any function:

```
sqrt(9);
sqrt(100 + 37);
sqrt(sqrt(2) + sqrt(3));
square(sqrt(1000));
```

The `sqrt` program also illustrates that the simple functional language we have introduced so far is sufficient for writing any purely numerical program that one could write in, say, C or Pascal. This might seem surprising, since we have not any iterative (looping) constructs that direct the computer to do something over and over again. The function `sqrt_iter`, on the other hand, demonstrates how iteration can be accomplished using no special construct other than the ordinary ability to call a .

Exercise 1.6

Alyssa P. Hacker doesn’t like the syntax of conditional expressions, involving the characters `? and :`. ‘Why can’t I just declare an ordinary conditional function whose application works

just like conditional expressions?’ she asks. Alyssa’s friend Eva Lu Ator claims this can indeed be done, and she declares a conditional function as follows:

```
function conditional(predicate, then_clause, else_clause) {
    return predicate ? then_clause : else_clause;
}
```

Eva demonstrates the program for Alyssa:

```
conditional(2 === 3, 0, 5);
```

evaluates as expected to 5, and

```
conditional(1 === 1, 0, 5);
```

evaluates as expected to 0. Delighted, Alyssa uses conditional to rewrite the square-root program:

```
function sqrt_iter(guess, x) {
    return conditional(good_enough(guess, x),
                      guess,
                      sqrt_iter(improve(guess, x),
                                x));
}
```

What happens when Alyssa attempts to use this to compute square roots? Explain. [Solution](#)

Exercise 1.7

The good_enough test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing good_enough is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root function that uses this kind of end test. Does this work better for small and large numbers? [Solution](#)

Exercise 1.8

Newton’s method for cube roots is based on the fact that if y is an approximation to the cube root of x , then a better approximation is given by the value

$$\frac{x/y^2 + 2y}{3}$$

Use this formula to implement a cube-root function analogous to the square-root function. (In section 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root functions.)

[Solution](#)

1.1.8 Functions as Black-Box Abstractions

The function `sqrt` is our first example of a process defined by a set of mutually defined functions. Notice that the declaration of `sqrt_iter` is *recursive*; that is, the function is defined in terms of itself. The idea of being able to define a function in terms of itself may be disturbing; it may seem unclear how such a ‘circular’ definition could make sense at all, much less specify a well-defined process to be carried out by a computer. This will be addressed more carefully in section 1.2. But first let’s consider some other important points illustrated by the `sqrt` example.

Observe that the problem of computing square roots breaks up naturally into a number of subproblems: how to tell whether a guess is good enough, how to improve a guess, and so on. Each of these tasks is accomplished by a separate function. The entire `sqrt` program can be viewed as a cluster of functions (shown in Figure 1.2) that mirrors the decomposition of the problem into subproblems.

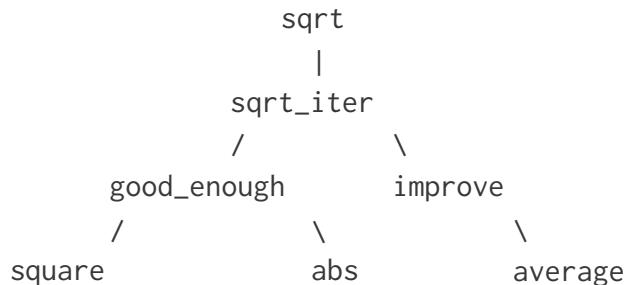


Figure 1.2: Functional decomposition of the `sqrt` program.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts—the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each function accomplishes an identifiable task that can be used as a module in defining other functions. For example, when we define the `good_enough` function in terms of `square`, we are able to regard the `square` function as a ‘black box.’ We are not at that moment concerned with *how* the function computes its result, only with the fact *that* it computes the square. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as the `good_enough` function is concerned, `square` is not quite a function but rather an abstraction of a function, a so-called *functional abstraction*. At this level of abstraction, any function that computes the square is equally good.

Thus, considering only the values they return, the following two functions squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.¹⁴

```
function square(x) {
    return x * x;
}

function square(x) {
    return math_exp(double(math_log(x)));
}
function double(x) {
    return x + x;
}
```

So a function should be able to suppress detail. The users of the function may not have written the function themselves, but may have obtained it from another programmer as a black box. A user should not need to know how the function is implemented in order to use it.

Local names

One detail of a function's implementation that should not matter to the user of the function is the implementer's choice of names for the function's parameters. Thus, the following functions should not be distinguishable:

```
function square(x) {
    return x * x;
}

function square(y) {
    return y * y;
}
```

This principle—that the meaning of a function should be independent of the parameter names used by its author—seems on the surface to be self-evident, but its consequences are profound. The simplest consequence is that the parameter names of a function must be local to the body of the function. For example, we used `square` in the declaration of `good_enough` in our square-root function :

```
function good_enough(guess, x) {
    return abs(square(guess) - x) < 0.001;
}
```

The intention of the author of `good_enough` is to determine if the square of the first argument is within a given tolerance of the second argument. We see that the author of `good_enough`

¹⁴It is not even clear which of these functions is a more efficient implementation. This depends upon the hardware available. There are machines for which the ‘obvious’ implementation is the less efficient one. Consider a machine that has extensive tables of logarithms and antilogarithms stored in a very efficient manner.

used the name `guess` to refer to the first argument and `x` to refer to the second argument. The argument of `square` is `guess`. If the author of `square` used `x` (as above) to refer to that argument, we see that the `x` in `good_enough` must be a different `x` than the one in `square`. Running the function `square` must not affect the value of `x` that is used by `good_enough`, because that value of `x` may be needed by `good_enough` after `square` is done computing.

If the parameters were not local to the bodies of their respective functions, then the parameter `x` in `square` could be confused with the parameter `x` in `good_enough`, and the behavior of `good_enough` would depend upon which version of `square` we used. Thus, `square` would not be the black box we desired.

A parameter of a function has a very special role in the function declaration, in that it doesn't matter what name the parameter has. Such a name is called *bound*, and we say that the function declaration *binds* its parameters. The meaning of a function declaration is unchanged if a bound name is consistently renamed throughout the declaration.¹⁵ If a name is not bound, we say that it is *free*. The set of expressions for which a binding declares a name is called the *scope* of that name. In a function declaration, the bound names declared as the parameters of the function have the body of the function as their scope.

In the declaration of `good_enough` above, `guess` and `x` are bound names but `abs`, and `square` are free. The meaning of `good_enough` should be independent of the names we choose for `guess` and `x` so long as they are distinct and different from `abs`, and `square`. (If we renamed `guess` to `abs` we would have introduced a bug by *capturing* the name `abs`. It would have changed from free to bound.) The meaning of `good_enough` is not independent of the choice of its free names, however. It surely depends upon the fact (external to this declaration) that the symbol `abs` names a function for computing the absolute value of a number. The JavaScript function `good_enough` will compute a different mathematical function if we substitute `math_cos` (JavaScript's cosine function) for `abs` in its declaration.

Internal declarations and block structure

We have one kind of name isolation available to us so far: The parameters of a function are local to the body of the function. The square-root program illustrates another way in which we would like to control the use of names. The existing program consists of separate functions :

¹⁵The concept of consistent renaming is actually subtle and difficult to define formally. Famous logicians have made embarrassing errors here.

```

function sqrt(x) {
    return sqrt_iter(1.0, x);
}
function sqrt_iter(guess, x) {
    return good_enough(guess, x)
        ? guess
        : sqrt_iter(improve(guess, x), x);
}
function good_enough(guess, x) {
    return abs(square(guess) - x) < 0.001;
}
function improve(guess, x) {
    return average(guess, x / guess);
}

```

The problem with this program is that the only function that is important to users of `sqrt` is `sqrt`. The other functions (`sqrt_iter`, `good_enough`, and `improve`) only clutter up their minds. They may not declare any other function called `good_enough` as part of another program to work together with the square-root program, because `sqrt` needs it. The problem is especially severe in the construction of large systems by many separate programmers. For example, in the construction of a large library of numerical functions, many numerical functions are computed as successive approximations and thus might have functions named `good_enough` and `improve` as auxiliary functions. We would like to localize the subfunctions, hiding them inside `sqrt` so that `sqrt` could coexist with other successive approximations, each having its own private `good_enough` function. To make this possible, we allow a function to have internal declarations that are local to that function. For example, in the square-root problem we can write

```

function sqrt(x) {
    function good_enough(guess, x) {
        return abs(square(guess) - x) < 0.001;
    }
    function improve(guess, x) {
        return average(guess, x / guess);
    }
    function sqrt_iter(guess, x) {
        return good_enough(guess, x)
            ? guess
            : sqrt_iter(improve(guess, x), x);
    }
    return sqrt_iter(1.0, x);
}

```

The body of a function—a statement enclosed in curly braces—is called a *block*. Function declarations nested inside a block are local to that block. This *block structure* is basically the right solution to the simplest name-packaging problem. But there is a better idea lurking

here. In addition to internalizing the declarations of the auxiliary functions, we can simplify them. Since `x` is bound in the declaration of `sqrt`, the functions `good_enough`, `improve`, and `sqrt_iter`, which are defined internally to `sqrt`, are in the scope of `x`. Thus, it is not necessary to pass `x` explicitly to each of these functions. Instead, we allow `x` to be a free name in the internal declarations, as shown below. Then `x` gets its value from the argument with which the enclosing function `sqrt` is called. This discipline is called *lexical scoping*.¹⁶

```
function sqrt(x) {
    function good_enough(guess) {
        return abs(square(guess) - x) < 0.001;
    }
    function improve(guess) {
        return average(guess, x / guess);
    }
    function sqrt_iter(guess) {
        return good_enough(guess)
            ? guess
            : sqrt_iter(improve(guess));
    }
    return sqrt_iter(1.0);
}
```

We will use block structure extensively to help us break up large programs into tractable pieces.¹⁷ The idea of block structure originated with the programming language Algol 60. It appears in most advanced programming languages and is an important tool for helping to organize the construction of large programs. We will see in the next section that function declarations can contain blocks other than function declarations. JavaScript's adherence to block structure is incomplete in that function declarations are local to the surrounding function declarations, not to the surrounding block. We shall therefore always place local function declarations first in the body of any function declaration.

¹⁶Lexical scoping dictates that free names in a function are taken to refer to bindings made by enclosing function declarations; that is, they are looked up in the environment in which the function was declared. We will see how this works in detail in chapter 3 when we study environments and the detailed behavior of the interpreter.

¹⁷Embedded declarations must come first in a function body. The management is not responsible for the consequences of running programs that intertwine declaration and use.

1.2 Functions and the Processes They Generate

We have now considered the elements of programming: We have used primitive arithmetic operations, we have combined these operations, and we have abstracted these composite operations by declaring them as compound functions. But that is not enough to enable us to say that we know how to program. Our situation is analogous to that of someone who has learned the rules for how the pieces move in chess but knows nothing of typical openings, tactics, or strategy. Like the novice chess player, we don't yet know the common patterns of usage in the domain. We lack the knowledge of which moves are worth making (which functions are worth declaring). We lack the experience to predict the consequences of making a move (executing a function).

The ability to visualize the consequences of the actions under consideration is crucial to becoming an expert programmer, just as it is in any synthetic, creative activity. In becoming an expert photographer, for example, one must learn how to look at a scene and know how dark each region will appear on a print for each possible choice of exposure and development¹⁸ conditions. Only then can one reason backward, planning framing, lighting, exposure, and development to obtain the desired effects. So it is with programming, where we are planning the course of action to be taken by a process and where we control the process by means of a program. To become experts, we must learn to visualize the processes generated by various types of functions. Only after we have developed such a skill can we learn to reliably construct programs that exhibit the desired behavior.

A function is a pattern for the *local evolution* of a computational process. It specifies how each stage of the process is built upon the previous stage. We would like to be able to make statements about the overall, or *global*, behavior of a process whose local evolution has been specified by a function. This is very difficult to do in general, but we can at least try to describe some typical patterns of process evolution.

In this section we will examine some common ‘shapes’ for processes generated by simple functions. We will also investigate the rates at which these processes consume the important computational resources of time and space. The functions we will consider are very simple. Their role is like that played by test patterns in photography: as oversimplified prototypical patterns, rather than practical examples in their own right.

¹⁸The textbook was written at a time when photography commonly involved photographic development, a chemical process for making paper prints from photographic film.

1.2.1 Linear Recursion and Iteration

We begin by considering the factorial function, defined by

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1$$

There are many ways to compute factorials. One way is to make use of the observation that $n!$ is equal to n times $(n - 1)!$ for any positive integer n :

$$n! = n \cdot [(n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!$$

Thus, we can compute $n!$ by computing $(n - 1)!$ and multiplying the result by n . If we add the stipulation that $1!$ is equal to 1, this observation translates directly into a function:

```
function factorial(n) {
    return n === 1
        ? 1
        : n * factorial(n - 1);
}
```

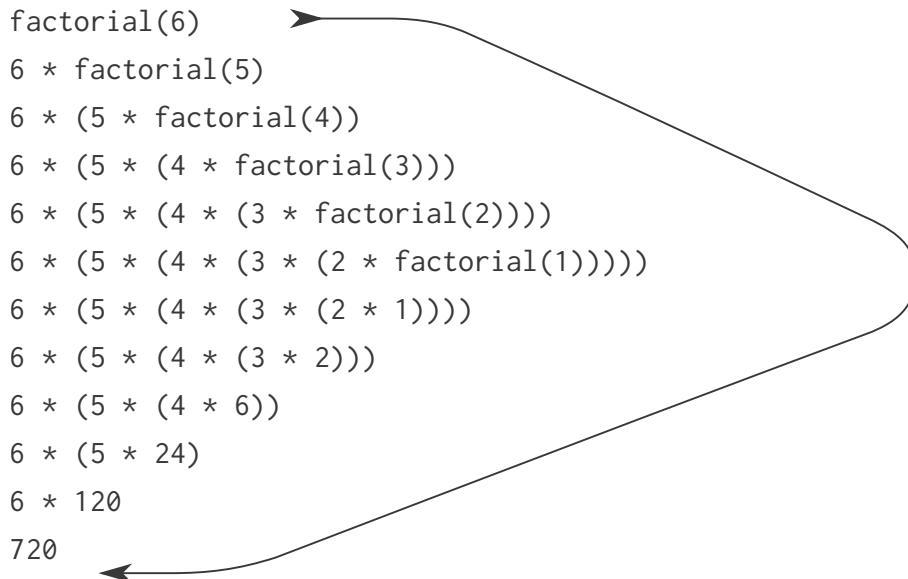


Figure 1.3: A linear recursive process for computing $6!$.

We can use the substitution model of section 1.1.5 to watch the function in action computing $6!$, as shown in figure 1.3.

Now let's take a different perspective on computing factorials. We could describe a rule for computing $n!$ by specifying that we first multiply 1 by 2, then multiply the result by 3, then by 4, and so on until we reach n . More formally, we maintain a running product, together with a counter that counts from 1 up to n . We can describe the computation by saying that the

counter and the product simultaneously change from one step to the next according to the rule

$\text{product} \leftarrow \text{counter} \cdot \text{product}$

$\text{counter} \leftarrow \text{counter} + 1$ and stipulating that $n!$ is the value of the product when the counter exceeds n .

Once again, we can recast our description as a function for computing factorials:¹⁹

```
function factorial(n) {
    return fact_iter(1, 1, n);
}
function fact_iter(product, counter, max_count) {
    return counter > max_count
        ? product
        : fact_iter(counter * product,
                    counter + 1,
                    max_count);
}
```

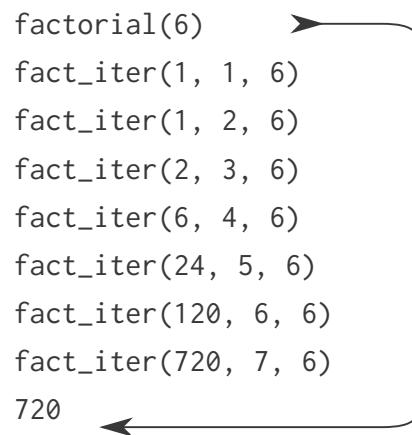


Figure 1.4: A linear iterative process for computing $6!$.

As before, we can use the substitution model to visualize the process of computing $6!$, as shown in figure 1.4.

¹⁹In a real program we would probably use the block structure introduced in the last section to hide the declaration of `fact_iter`:

```
function factorial(n) {
    function iter(product, counter) {
        return counter > n
            ? product
            : iter(counter * product,
                  counter + 1);
    }
    return iter(1, 1);
}
```

We avoided doing this here so as to minimize the number of things to think about at once.

Compare the two processes. From one point of view, they seem hardly different at all. Both compute the same mathematical function on the same domain, and each requires a number of steps proportional to n to compute $n!$. Indeed, both processes even carry out the same sequence of multiplications, obtaining the same sequence of partial products. On the other hand, when we consider the ‘shapes’ of the two processes, we find that they evolve quite differently.

Consider the first process. The substitution model reveals a shape of expansion followed by contraction, indicated by the arrow in figure 1.3. The expansion occurs as the process builds up a chain of *deferred operations* (in this case, a chain of multiplications). The contraction occurs as the operations are actually performed. This type of process, characterized by a chain of deferred operations, is called a *recursive process*. Carrying out this process requires that the interpreter keep track of the operations to be performed later on. In the computation of $n!$, the length of the chain of deferred multiplications, and hence the amount of information needed to keep track of it, grows linearly with n (is proportional to n), just like the number of steps. Such a process is called a *linear recursive process*.

By contrast, the second process does not grow and shrink. At each step, all we need to keep track of, for any n , are the current values of the names `product`, `counter`, and `max_count`. We call this an *iterative process*. In general, an iterative process is one whose state can be summarized by the values of a fixed number of *state names*, together with a fixed rule that describes how the values of the state names should be updated as the process moves from state to state and an (optional) end test that specifies conditions under which the process should terminate. In computing $n!$, the number of steps required grows linearly with n . Such a process is called a *linear iterative process*.

The contrast between the two processes can be seen in another way. In the iterative case, the values of the state names provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three state names. Not so with the recursive process. In this case there is some additional ‘hidden’ information, maintained by the interpreter and not contained in the state names, which indicates ‘where the process is’ in negotiating the chain of deferred operations. The longer the chain, the more information must be maintained.²⁰

In contrasting iteration and recursion, we must be careful not to confuse the notion of a recursive *process* with the notion of a recursive *function*. When we describe a function as recursive, we are referring to the syntactic fact that the function declaration refers (either directly or indirectly) to the function itself. But when we describe a process as following a

²⁰When we discuss the implementation of functions on register machines in chapter 5, we will see that any iterative process can be realized ‘in hardware’ as a machine that has a fixed set of registers and no auxiliary memory. In contrast, realizing a recursive process requires a machine that uses an auxiliary data structure known as a *stack*.

pattern that is, say, linearly recursive, we are speaking about how the process evolves, not about the syntax of how a function is written. It may seem disturbing that we refer to a recursive function such as `fact_iter` as generating an iterative process. However, the process really is iterative: Its state is captured completely by its three state names, and an interpreter need keep track of only three names in order to execute the process.

One reason that the distinction between process and procedure may be confusing is that most implementations of common languages (including Ada, Pascal, and C) are designed in such a way that the interpretation of any recursive function consumes an amount of memory that grows with the number of function calls, even when the process described is, in principle, iterative. As a consequence, these languages can describe iterative processes only by resorting to special-purpose ‘looping constructs’ such as `do`, `repeat`, `until`, `for`, and `while`. The implementation of JavaScript we shall consider in chapter 5 does not share this defect. It will execute an iterative process in constant space, even if the iterative process is described by a recursive function. An implementation with this property is called *tail-recursive*. With a tail-recursive implementation, iteration can be expressed using the ordinary function call mechanism, so that special iteration constructs are useful only as syntactic sugar.²¹

Exercise 1.9

Each of the following two functions defines a method for adding two positive integers in terms of the functions `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
function plus(a, b) {
    return a === 0 ? b : inc(plus(dec(a), b));
}

function plus(a, b) {
    return a === 0 ? b : plus(dec(a), inc(b));
}
```

Using the substitution model, illustrate the process generated by each function in evaluating `plus(4, 5);`. Are these processes iterative or recursive?

[Solution](#)

²¹Tail recursion has long been known as a compiler optimization trick. A coherent semantic basis for tail recursion was provided by Carl Hewitt (1977), who explained it in terms of the ‘message-passing’ model of computation that we shall discuss in chapter 3. Inspired by this, Gerald Jay Sussman and Guy Lewis Steele Jr. (see Steele 1975) constructed a tail-recursive interpreter for Scheme. Steele later showed how tail recursion is a consequence of the natural way to compile function calls (Steele 1977). The IEEE standard for Scheme requires that Scheme implementations be tail-recursive. The ECMA standard for JavaScript eventually followed suit with ECMAScript 2015 (ECMA 2015). Note however, that as of this writing (2019), most implementations of JavaScript do not comply with this standard.

Exercise 1.10

The following function computes a mathematical function called Ackermann's function.

```
function A(x, y) {
    return y === 0
        ? 0
        : x === 0
            ? 2 * y
            : y === 1
                ? 2
                : A(x - 1, A(x, y - 1));
}
```

What are the values of the following expressions?

A(1, 10);

A(2, 4);

A(3, 3);

Consider the following functions, where A is the function declared above:

```
function f(n) {
    return A(0, n);
}
function g(n) {
    return A(1, n);
}
function h(n) {
    return A(2, n);
}
function k(n) {
    return 5 * n * n;
}
```

Give concise mathematical definitions for the functions computed by the functions f, g, and h for positive integer values of n . For example, $k(n)$ computes $5n^2$.

[Solution](#)

1.2.2 Tree Recursion

Another common pattern of computation is called *tree recursion*. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

In general, the Fibonacci numbers can be defined by the rule

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{otherwise} \end{cases}$$

We can immediately translate this definition into a recursive function for computing Fibonacci numbers:

```
function fib(n) {
    return n === 0
        ? 0
        : n === 1
        ? 1
        : fib(n - 1) + fib(n - 2);
}
```

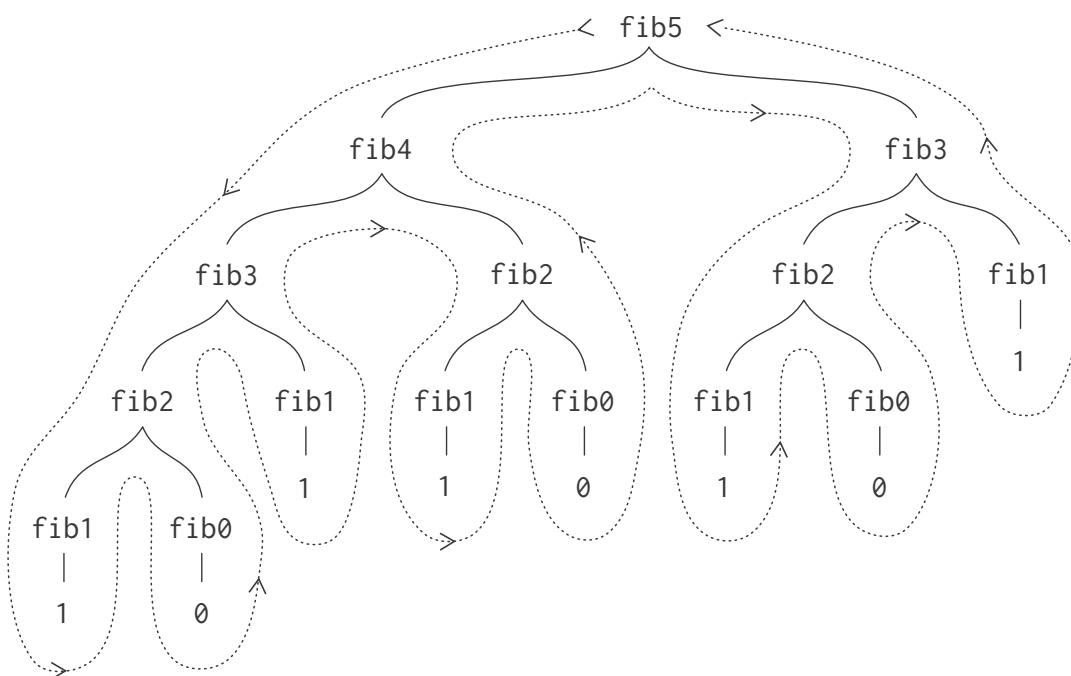


Figure 1.5: The tree-recursive process generated in computing $\text{fib}(5)$.

Consider the pattern of this computation. To compute $\text{fib}(5)$, we compute $\text{fib}(4)$ and $\text{fib}(3)$

. To compute `fib(4)` , we compute `fib(3)` and `fib(2)` . In general, the evolved process looks like a tree, as shown in figure 1.5. Notice that the branches split into two at each level (except at the bottom); this reflects the fact that the `fib` function calls itself twice each time it is invoked.

This function is instructive as a prototypical tree recursion, but it is a terrible way to compute Fibonacci numbers because it does so much redundant computation. Notice in figure 1.5 that the entire computation of `fib(3)`—almost half the work—is duplicated. In fact, it is not hard to show that the number of times the function will compute `fib(1)` or `fib(0)` (the number of leaves in the above tree, in general) is precisely $\text{Fib}(n + 1)$. To get an idea of how bad this is, one can show that the value of $\text{Fib}(n)$ grows exponentially with n . More precisely (see exercise 1.13), $\text{Fib}(n)$ is the closest integer to $\phi^n/\sqrt{5}$, where

$$\phi = (1 + \sqrt{5})/2 \approx 1.6180$$

is the *golden ratio*, which satisfies the equation

$$\phi^2 = \phi + 1$$

Thus, the process uses a number of steps that grows exponentially with the input. On the other hand, the space required grows only linearly with the input, because we need keep track only of which nodes are above us in the tree at any point in the computation. In general, the number of steps required by a tree-recursive process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree.

We can also formulate an iterative process for computing the Fibonacci numbers. The idea is to use a pair of integers a and b , initialized to $\text{Fib}(1) = 1$ and $\text{Fib}(0) = 0$, and to repeatedly apply the simultaneous transformations

$$\begin{aligned} a &\leftarrow a + b \\ b &\leftarrow a \end{aligned}$$

It is not hard to show that, after applying this transformation n times, a and b will be equal, respectively, to $\text{Fib}(n + 1)$ and $\text{Fib}(n)$. Thus, we can compute Fibonacci numbers iteratively using the function

```
function fib(n) {
    return fib_iter(1, 0, n);
}
function fib_iter(a, b, count) {
    return count === 0
        ? b
        : fib_iter(a + b, a, count - 1);
```

```
}
```

This second method for computing $\text{Fib}(n)$ is a linear iteration. The difference in number of steps required by the two methods—one linear in n , one growing as fast as $\text{Fib}(n)$ itself—is enormous, even for small inputs.

One should not conclude from this that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool.²² But even in numerical operations, tree-recursive processes can be useful in helping us to understand and design programs. For instance, although the first `fib` function is much less efficient than the second one, it is more straightforward, being little more than a translation into JavaScript of the definition of the Fibonacci sequence. To formulate the iterative algorithm required noticing that the computation could be recast as an iteration with three state names.

Example: Counting change

It takes only a bit of cleverness to come up with the iterative Fibonacci algorithm. In contrast, consider the following problem: How many different ways can we make change of \$1.00, given half-dollars, quarters, dimes, nickels, and pennies? More generally, can we write a function to compute the number of ways to change any given amount of money?

This problem has a simple solution as a recursive function. Suppose we think of the types of coins available as arranged in some order. Then the following relation holds: The number of ways to change amount a using n kinds of coins equals

- the number of ways to change amount a using all but the first kind of coin, plus
- the number of ways to change amount $a - d$ using all n kinds of coins, where d is the denomination of the first kind of coin.

To see why this is true, observe that the ways to make change can be divided into two groups: those that do not use any of the first kind of coin, and those that do. Therefore, the total number of ways to make change for some amount is equal to the number of ways to make change for the amount without using any of the first kind of coin, plus the number of ways to make change assuming that we do use the first kind of coin. But the latter number is equal to the number of ways to make change for the amount that remains after using a coin of the first kind.

Thus, we can recursively reduce the problem of changing a given amount to problems of changing smaller amounts or using fewer kinds of coins. Consider this reduction rule carefully,

²²An example of this was hinted at in section 1.1.3: The interpreter itself evaluates expressions using a tree-recursive process.

and convince yourself that we can use it to describe an algorithm if we specify the following degenerate cases:²³

- If a is exactly 0, we should count that as 1 way to make change.
- If a is less than 0, we should count that as 0 ways to make change.
- If n is 0, we should count that as 0 ways to make change.

We can easily translate this description into a recursive function:

```
function count_change(amount) {
    return cc(amount, 5);
}
function cc(amount, kinds_of_coins) {
    return amount === 0
        ? 1
        : amount < 0 || kinds_of_coins === 0
        ? 0
        : cc(amount, kinds_of_coins - 1)
        +
        cc(amount - first_denomination(
            kinds_of_coins),
            kinds_of_coins);
}
function first_denomination(kinds_of_coins) {
    return kinds_of_coins === 1 ? 1 :
        kinds_of_coins === 2 ? 5 :
        kinds_of_coins === 3 ? 10 :
        kinds_of_coins === 4 ? 25 :
        kinds_of_coins === 5 ? 50 : 0;
}
```

(The `first_denomination` function takes as input the number of kinds of coins available and returns the denomination of the first kind. Here we are thinking of the coins as arranged in order from largest to smallest, but any order would do as well.) We can now answer our original question about changing a dollar:

```
count_change(100);
```

The function `count_change` generates a tree-recursive process with redundancies similar to those in our first implementation of `fib`. (It will take quite a while for that 292 to be computed.) On the other hand, it is not obvious how to design a better algorithm for computing the result,

²³For example, work through in detail how the reduction rule applies to the problem of making change for 10 cents using pennies and nickels.

and we leave this problem as a challenge. The observation that a tree-recursive process may be highly inefficient but often easy to specify and understand has led people to propose that one could get the best of both worlds by designing a ‘smart compiler’ that could transform tree-recursive functions into more efficient functions that compute the same result.²⁴

Exercise 1.11

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n - 1) + 2f(n - 2) + 3f(n - 3)$ if $n \geq 3$. Write a JavaScript function that computes f by means of a recursive process. Write a function that computes f by means of an iterative process.

[Solution](#)

Exercise 1.12

The following pattern of numbers is called *Pascal’s triangle*.

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.²⁵ Write a function that computes elements of Pascal’s triangle by means of a recursive process.

[Solution](#)

Exercise 1.13

Prove that $\text{Fib}(n)$ is the closest integer to $\phi^n/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2$. Hint: Let $\psi = (1-\sqrt{5})/2$. Use induction and the definition of the Fibonacci numbers (see section 1.2.2) to prove that

²⁴One approach to coping with redundant computations is to arrange matters so that we automatically construct a table of values as they are computed. Each time we are asked to apply the function to some argument, we first look to see if the value is already stored in the table, in which case we avoid performing the redundant computation. This strategy, known as *tabulation* or *memoization*, can be implemented in a straightforward way. Tabulation can sometimes be used to transform processes that require an exponential number of steps (such as count-change) into processes whose space and time requirements grow linearly with the input. See exercise 3.27.

²⁵The elements of Pascal’s triangle are called the *binomial coefficients*, because the n th row consists of the coefficients of the terms in the expansion of $(x+y)^n$. This pattern for computing the coefficients appeared in Blaise Pascal’s 1653 seminal work on probability theory, *Traité du triangle arithmétique*. According to Knuth (1973), the same pattern appears in the *Szu-yuen Yü-chien* (‘The Precious Mirror of the Four Elements’), published by the Chinese mathematician Chu Shih-chieh in 1303, in the works of the twelfth-century Persian poet and mathematician Omar Khayyam, and in the works of the twelfth-century Hindu mathematician Bháskara Áchárya.

$$\text{Fib}(n) = (\phi^n - \psi^n)/\sqrt{5}.$$

1.2.3 Orders of Growth

The previous examples illustrate that processes can differ considerably in the rates at which they consume computational resources. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a gross measure of the resources required by a process as the inputs become larger.

Let n be a parameter that measures the size of the problem, and let $R(n)$ be the amount of resources the process requires for a problem of size n . In our previous examples we took n to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take n to be the number of digits accuracy required. For matrix multiplication we might take n to be the number of rows in the matrices. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly, $R(n)$ might measure the number of internal storage registers used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required will be proportional to the number of elementary machine operations performed.

We say that $R(n)$ has order of growth $\Theta(f(n))$, written $R(n) = \Theta(f(n))$ (pronounced ‘theta of $f(n)$ ’), if there are positive constants k_1 and k_2 independent of n such that $k_1f(n) \leq R(n) \leq k_2f(n)$ for any sufficiently large value of n . (In other words, for large n , the value $R(n)$ is sandwiched between $k_1f(n)$ and $k_2f(n)$.)

For instance, with the linear recursive process for computing factorial described in section 1.2.1 the number of steps grows proportionally to the input n . Thus, the steps required for this process grows as $\Theta(n)$. We also saw that the space required grows as $\Theta(n)$. For the iterative factorial, the number of steps is still $\Theta(n)$ but the space is $\Theta(1)$ —that is, constant.²⁶ The tree-recursive Fibonacci computation requires $\Theta(\phi^n)$ steps and space $\Theta(n)$, where ϕ is the golden ratio described in section 1.2.2.

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n^2 steps and a process requiring $1000n^2$ steps and a process requiring $3n^2 + 10n + 17$ steps all have $\Theta(n^2)$ order of growth. On the other hand, order of growth provides a useful indication of how we may expect the behavior of the process to change as

²⁶These statements mask a great deal of oversimplification. For instance, if we count process steps as ‘machine operations’ we are making the assumption that the number of machine operations needed to perform, say, a multiplication is independent of the size of the numbers to be multiplied, which is false if the numbers are sufficiently large. Similar remarks hold for the estimates of space. Like the design and description of a process, the analysis of a process can be carried out at various levels of abstraction.

we change the size of the problem. For a $\Theta(n)$ (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. In the remainder of section 1.2 we will examine two algorithms whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by a constant amount.

Exercise 1.14

Draw the tree illustrating the process generated by the `count_change` function of section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

[Solution](#)

Exercise 1.15

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ if x is sufficiently small, and the trigonometric identity $\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$ to reduce the size of the argument of \sin . (For purposes of this exercise an angle is considered ‘sufficiently small’ if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following functions:

```
function cube(x) {
    return x * x * x;
}
function p(x) {
    return 3 * x - 4 * cube(x);
}
function sine(angle) {
    return !(abs(angle) > 0.1)
        ? angle
        : p(sine(angle / 3.0));
}
```

- How many times is the function `p` applied when `sine(12.15)` is evaluated?
- What is the order of growth in space and number of steps (as a function of a) used by the process generated by the `sine` function when `sine(a)` is evaluated?

[Solution](#)

1.2.4 Exponentiation

Consider the problem of computing the exponential of a given number. We would like a function that takes as arguments a base b and a positive integer exponent n and computes b^n . One way to do this is via the recursive definition

$$\begin{aligned} b^n &= b \cdot b^{n-1} \\ b^0 &= 1 \end{aligned}$$

which translates readily into the function

```
function expt(b,n) {
    return n === 0
        ? 1
        : b * expt(b, n - 1);
}
```

This is a linear recursive process, which requires $\Theta(n)$ steps and $\Theta(n)$ space. Just as with factorial, we can readily formulate an equivalent linear iteration:

```
function expt(b,n) {
    return expt_iter(b,n,1);
}
function expt_iter(b,counter,product) {
    return counter === 0
        ? product
        : expt_iter(b,
                    counter - 1,
                    b * product);
}
```

This version requires $\Theta(n)$ steps and $\Theta(1)$ space.

We can compute exponentials in fewer steps by using successive squaring. For instance, rather than computing b^8 as

$$b \cdot (b \cdot b)))))))$$

we can compute it using three multiplications:

$$\begin{aligned} b^2 &= b \cdot b \\ b^4 &= b^2 \cdot b^2 \\ b^8 &= b^4 \cdot b^4 \end{aligned}$$

This method works fine for exponents that are powers of 2. We can also take advantage of

successive squaring in computing exponentials in general if we use the rule

$$\begin{aligned} b^n &= (b^{n/2})^2 && \text{if } n \text{ is even} \\ b^n &= b \cdot b^{n-1} && \text{if } n \text{ is odd} \end{aligned}$$

We can express this method as a function:

```
function fast_expt(b, n) {
    return n === 0
        ? 1
        : is_even(n)
            ? square(fast_expt(b, n / 2))
            : b * fast_expt(b, n - 1);
}
```

where the predicate to test whether an integer is even is defined in terms of the operator %, which computes the remainder after integer division, by

```
function is_even(n) {
    return n % 2 === 0;
}
```

The process evolved by `fast_expt` grows logarithmically with n in both space and number of steps. To see this, observe that computing b^{2n} using `fast_expt` requires only one more multiplication than computing b^n . The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed. Thus, the number of multiplications required for an exponent of n grows about as fast as the logarithm of n to the base 2. The process has $\Theta(\log n)$ growth.²⁷

The difference between $\Theta(\log n)$ growth and $\Theta(n)$ growth becomes striking as n becomes large. For example, `fast_expt` for $n = 1000$ requires only 14 multiplications.²⁸ It is also possible to use the idea of successive squaring to devise an iterative algorithm that computes exponentials with a logarithmic number of steps (see exercise 1.16), although, as is often the case with iterative algorithms, this is not written down so straightforwardly as the recursive algorithm.²⁹

Exercise 1.16

Design a function that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast_expt`. (Hint: Using the observation

²⁷More precisely, the number of multiplications required is equal to 1 less than the log base 2 of n , plus the number of ones in the binary representation of n . This total is always less than twice the log base 2 of n . The arbitrary constants k_1 and k_2 in the definition of order notation imply that, for a logarithmic process, the base to which logarithms are taken does not matter, so all such processes are described as $\Theta(\log n)$.

²⁸You may wonder why anyone would care about raising numbers to the 1000th power. See section 1.2.6.

²⁹This iterative algorithm is ancient. It appears in the *Chandah-sutra* by Áchárya, written before 200 B.C. See Knuth 1981, section 4.6.3, for a full discussion and analysis of this and other methods of exponentiation.

that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b , an additional state name a , and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

[Solution](#)

Exercise 1.17

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication function (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` function:

```
function times(a,b) {
    return b === 0
        ? 0
        : a + times(a, b - 1);
}
```

This algorithm takes a number of steps that is linear in b . Now suppose we include, together with addition, operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication function analogous to `fast_expt` that uses a logarithmic number of steps.

[Solution](#)

Exercise 1.18

Using the results of exercises 1.16 and 1.17, devise a function that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.³⁰

[Solution](#)

Exercise 1.19

There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state names a and b in the `fib_iter` process of section 1.2.2: $a \leftarrow a + b$ and $b \leftarrow a$. Call this transformation T , and observe that applying

³⁰This algorithm, which is sometimes known as the ‘Russian peasant method’ of multiplication, is ancient. Examples of its use are found in the Rhind Papyrus, one of the two oldest mathematical documents in existence, written about 1700 B.C. (and copied from an even older document) by an Egyptian scribe named A'h-mose.

T over and over again n times, starting with 1 and 0, produces the pair $\text{Fib}(n + 1)$ and $\text{Fib}(n)$. In other words, the Fibonacci numbers are produced by applying T^n , the n th power of the transformation T , starting with the pair $(1, 0)$. Now consider T to be the special case of $p = 0$ and $q = 1$ in a family of transformations T_{pq} , where T_{pq} transforms the pair (a, b) according to $a \leftarrow bq + aq + ap$ and $b \leftarrow bp + aq$. Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q . This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the `fast_expt` function. Put this all together to complete the following function, which runs in a logarithmic number of steps:³¹

```

function fib(n) {
    return fib_iter(1, 0, 0, 1, n);
}
function fib_iter(a,b,p,q,count) {
    return count === 0
        ? b
        : is_even(count)
            ? fib_iter(a,
                        b,
                        ??,           // compute p'
                        ??,           // compute q'
                        count / 2)
            : fib_iter(b * q + a * q + a * p,
                        b * p + a * q,
                        p,
                        q,
                        count - 1);
}

```

[Solution](#)

1.2.5 Greatest Common Divisors

The greatest common divisor (GCD) of two integers a and b is defined to be the largest integer that divides both a and b with no remainder. For example, the GCD of 16 and 28 is 4. In chapter 2, when we investigate how to implement rational-number arithmetic, we will need to be able to compute GCDs in order to reduce rational numbers to lowest terms. (To reduce a rational number to lowest terms, we must divide both the numerator and the denominator by their GCD. For example, $16/28$ reduces to $4/7$.) One way to find the GCD of two integers is to factor them and search for common factors, but there is a famous algorithm that is much more efficient.

³¹This exercise was suggested to us by Joe Stoy, based on an example in Kaldewaij 1990.

The idea of the algorithm is based on the observation that, if r is the remainder when a is divided by b , then the common divisors of a and b are precisely the same as the common divisors of b and r . Thus, we can use the equation

$$\text{GCD}(a, b) = \text{GCD}(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$$\begin{aligned} \text{GCD}(206, 40) &= \text{GCD}(40, 6) \\ &= \text{GCD}(6, 4) \\ &= \text{GCD}(4, 2) \\ &= \text{GCD}(2, 0) \\ &= 2 \end{aligned}$$

reduces $\text{GCD}(206, 40)$ to $\text{GCD}(2, 0)$, which is 2. It is possible to show that starting with any two positive integers and performing repeated reductions will always eventually produce a pair where the second number is 0. Then the GCD is the other number in the pair. This method for computing the GCD is known as *Euclid's Algorithm*.³²

It is easy to express Euclid's Algorithm as a function:

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

This generates an iterative process, whose number of steps grows as the logarithm of the numbers involved.

The fact that the number of steps required by Euclid's Algorithm has logarithmic growth bears an interesting relation to the Fibonacci numbers:

Lamé's Theorem: If Euclid's Algorithm requires k steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the k th Fibonacci number.³³

³²Euclid's Algorithm is so called because it appears in Euclid's *Elements* (Book 7, ca. 300 b.c. According to Knuth (1973), it can be considered the oldest known nontrivial algorithm. The ancient Egyptian method of multiplication (exercise 1.18) is surely older, but, as Knuth explains, Euclid's algorithm is the oldest known to have been presented as a general algorithm, rather than as a set of illustrative examples.

³³This theorem was proved in 1845 by Gabriel Lamé, a French mathematician and engineer known chiefly for his contributions to mathematical physics. To prove the theorem, we consider pairs (a_k, b_k) , where $a_k \geq b_k$, for which Euclid's Algorithm terminates in k steps. The proof is based on the claim that, if $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ are three successive pairs in the reduction process, then we must have $b_{k+1} \geq b_k + b_{k-1}$. To verify the claim, consider that a reduction step is defined by applying the transformation $a_{k-1} = b_k$, $b_{k-1} = \text{remainder of } a_k \text{ divided by } b_k$. The second equation means that $a_k = qb_k + b_{k-1}$ for some positive integer q . And since q must be at least 1 we have $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$. But in the previous reduction step we have $b_{k+1} = a_k$. Therefore, $b_{k+1} = a_k \geq b_k + b_{k-1}$. This verifies the claim. Now we can prove the theorem by induction on k , the number of steps that the algorithm requires to terminate. The result is true for $k = 1$,

We can use this theorem to get an order-of-growth estimate for Euclid's Algorithm. Let n be the smaller of the two inputs to the function. If the process takes k steps, then we must have $n \geq \text{Fib}(k) \approx \phi^k / \sqrt{5}$. Therefore the number of steps k grows as the logarithm (to the base ϕ) of n . Hence, the order of growth is $\Theta(\log n)$.

Exercise 1.20

The process that a function generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative gcd function given above. Suppose we were to interpret this function using normal-order evaluation, as discussed in section 1.1.5. (The normal-order-evaluation rule for **if** is described in exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating $\text{gcd}(206, 40)$ and indicate the remainder operations that are actually performed. How many remainder operations are actually performed in the normal-order evaluation of $\text{gcd}(206, 40)$? In the applicative-order evaluation?

[Solution](#)

1.2.6 Example: Testing for Primality

This section describes two methods for checking the primality of an integer n , one with order of growth $\Theta(\sqrt{n})$, and a ‘probabilistic’ algorithm with order of growth $\Theta(\log n)$. The exercises at the end of this section suggest programming projects based on these algorithms.

Searching for divisors

Since ancient times, mathematicians have been fascinated by problems concerning prime numbers, and many people have worked on the problem of determining ways to test if numbers are prime. One way to test if a number is prime is to find the number’s divisors. The following program finds the smallest integral divisor (greater than 1) of a given number n . It does this in a straightforward way, by testing n for divisibility by successive integers starting with 2.

```
function smallest_divisor(n) {
    return find_divisor(n, 2);
}
function find_divisor(n, test_divisor) {
    return square(test_divisor) > n
        ? n
```

since this merely requires that b be at least as large as $\text{Fib}(1) = 1$. Now, assume that the result is true for all integers less than or equal to k and establish the result for $k + 1$. Let $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ be successive pairs in the reduction process. By our induction hypotheses, we have $b_{k-1} \geq \text{Fib}(k - 1)$ and $b_k \geq \text{Fib}(k)$. Thus, applying the claim we just proved together with the definition of the Fibonacci numbers gives $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k - 1) = \text{Fib}(k + 1)$, which completes the proof of Lamé’s Theorem.

```

        : divides(test_divisor, n)
        ? test_divisor
        : find_divisor(n, test_divisor + 1);
}
function divides(a, b) {
    return b % a === 0;
}

```

We can test whether a number is prime as follows: n is prime if and only if n is its own smallest divisor.

```

function is_prime(n) {
    return n === smallest_divisor(n);
}

```

The end test for `find_divisor` is based on the fact that if n is not prime it must have a divisor less than or equal to \sqrt{n} .³⁴ This means that the algorithm need only test divisors between 1 and \sqrt{n} . Consequently, the number of steps required to identify n as prime will have order of growth $\Theta(\sqrt{n})$.

The Fermat test

The $\Theta(\log n)$ primality test is based on a result from number theory known as Fermat's Little Theorem.³⁵

Fermat's Little Theorem: If n is a prime number and a is any positive integer less than n , then a raised to the n th power is congruent to a modulo n . (Two numbers are said to be *congruent modulo n* if they both have the same remainder when divided by n . The remainder of a number a when divided by n is also referred to as the *remainder of a modulo n*, or simply as *a modulo n*.)

If n is not prime, then, in general, most of the numbers $a < n$ will not satisfy the above relation. This leads to the following algorithm for testing primality: Given a number n , pick a random number $a < n$ and compute the remainder of a^n modulo n . If the result is not equal to a , then n is certainly not prime. If it is a , then chances are good that n is prime. Now pick another random number a and test it with the same method. If it also satisfies the equation, then we can be even more confident that n is prime. By trying more and more values of a , we can

³⁴If d is a divisor of n , then so is n/d . But d and n/d cannot both be greater than \sqrt{n} .

³⁵Pierre de Fermat (1601–1665) is considered to be the founder of modern number theory. He obtained many important number-theoretic results, but he usually announced just the results, without providing his proofs. Fermat's Little Theorem was stated in a letter he wrote in 1640. The first published proof was given by Euler in 1736 (and an earlier, identical proof was discovered in the unpublished manuscripts of Leibniz). The most famous of Fermat's results—known as Fermat's Last Theorem—was jotted down in 1637 in his copy of the book *Arithmetic* (by the third-century Greek mathematician Diophantus) with the remark 'I have discovered a truly remarkable proof, but this margin is too small to contain it.' Finding a proof of Fermat's Last Theorem became one of the most famous challenges in number theory. A complete solution was finally given in 1995 by Andrew Wiles of Princeton University.

increase our confidence in the result. This algorithm is known as the Fermat test.

To implement the Fermat test, we need a function that computes the exponential of a number modulo another number:

```
function expmod(base, exp, m) {
    return exp === 0
        ? 1
        : is_even(exp)
            ? square(expmod(base, exp / 2, m)) % m
            : (base * expmod(base, exp - 1, m)) % m;
}
```

This is very similar to the `fast_expt` function of section 1.2.4. It uses successive squaring, so that the number of steps grows logarithmically with the exponent.³⁶

The Fermat test is performed by choosing at random a number a between 1 and $n - 1$ inclusive and checking whether the remainder modulo n of the n th power of a is equal to a . The random number a is chosen using the function `random`, which we assume our JavaScript environment defines as a primitive function. The function `random` returns a nonnegative integer less than its integer input. Hence, to obtain a random number between 1 and $n - 1$, we call `random` with an input of $n - 1$ and add 1 to the result:

```
function fermat_test(n) {
    function try_it(a) {
        return expmod(a, n, n) === a;
    }
    return try_it(1 + random(n - 1));
}
```

The following function runs the test a given number of times, as specified by a parameter. Its value is true if the test succeeds every time, and false otherwise.

```
function fast_is_prime(n, times) {
    return times === 0
        ? true
        : fermat_test(n)
            ? fast_is_prime(n, times - 1)
            : false;
}
```

³⁶The reduction steps in the cases where the exponent e is greater than 1 are based on the fact that, for any integers x , y , and m , we can find the remainder of x times y modulo m by computing separately the remainders of x modulo m and y modulo m , multiplying these, and then taking the remainder of the result modulo m . For instance, in the case where e is even, we compute the remainder of $b^{e/2}$ modulo m , square this, and take the remainder modulo m . This technique is useful because it means we can perform our computation without ever having to deal with numbers much larger than m . (Compare exercise 1.25.)

Probabilistic methods

The Fermat test differs in character from most familiar algorithms, in which one computes an answer that is guaranteed to be correct. Here, the answer obtained is only probably correct. More precisely, if n ever fails the Fermat test, we can be certain that n is not prime. But the fact that n passes the test, while an extremely strong indication, is still not a guarantee that n is prime. What we would like to say is that for any number n , if we perform the test enough times and find that n always passes the test, then the probability of error in our primality test can be made as small as we like.

Unfortunately, this assertion is not quite correct. There do exist numbers that fool the Fermat test: numbers n that are not prime and yet have the property that a^n is congruent to a modulo n for all integers $a < n$. Such numbers are extremely rare, so the Fermat test is quite reliable in practice.³⁷

There are variations of the Fermat test that cannot be fooled. In these tests, as with the Fermat method, one tests the primality of an integer n by choosing a random integer $a < n$ and checking some condition that depends upon n and a . (See exercise 1.28 for an example of such a test.) On the other hand, in contrast to the Fermat test, one can prove that, for any n , the condition does not hold for most of the integers $a < n$ unless n is prime. Thus, if n passes the test for some random choice of a , the chances are better than even that n is prime. If n passes the test for two random choices of a , the chances are better than 3 out of 4 that n is prime. By running the test with more and more randomly chosen values of a we can make the probability of error as small as we like.

The existence of tests for which one can prove that the chance of error becomes arbitrarily small has sparked interest in algorithms of this type, which have come to be known as *probabilistic algorithms*. There is a great deal of research activity in this area, and probabilistic algorithms have been fruitfully applied to many fields.³⁸

Exercise 1.21

³⁷ Numbers that fool the Fermat test are called *Carmichael numbers*, and little is known about them other than that they are extremely rare. There are 255 Carmichael numbers below 100,000,000. The smallest few are 561, 1105, 1729, 2465, 2821, and 6601. In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a ‘correct’ algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

³⁸ One of the most striking applications of probabilistic prime testing has been to the field of cryptography. Although it is now computationally infeasible to factor an arbitrary 200-digit number, the primality of such a number can be checked in a few seconds with the Fermat test. This fact forms the basis of a technique for constructing ‘unbreakable codes’ suggested by Rivest, Shamir, and Adleman (1977). The resulting *RSA algorithm* has become a widely used technique for enhancing the security of electronic communications. Because of this and related developments, the study of prime numbers, once considered the epitome of a topic in ‘pure’ mathematics to be studied only for its own sake, now turns out to have important practical applications to cryptography, electronic funds transfer, and information retrieval.

Use the `smallest_divisor` function to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

[Solution](#)

Exercise 1.22

Assume that our JavaScript environment declares a primitive function called `runtime` that returns an integer that specifies the amount of time the system has been running (measured in microseconds). The following `timed_prime_test` function, when called with an integer n , prints n and checks to see if n is prime. If n is prime, the function prints three asterisks followed by the amount of time used in performing the test.³⁹

```
function timed_prime_test(n) {
    display(n);
    return start_prime_test(n, runtime());
}
function start_prime_test(n, start_time) {
    return is_prime(n)
        ? report_prime(runtime() - start_time)
        : true;
}
function report_prime(elapsed_time) {
    display(" *** ");
    display(elapsed_time);
}
```

Using this function, write a function `search_for_primes` that checks the primality of consecutive odd integers in a specified range. Use your function to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of $\Theta(\sqrt{n})$, you should expect that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the \sqrt{n} prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

Exercise 1.23

The `smallest_divisor` function shown at the start of this section does lots of needless testing:

³⁹Note that the function `display` takes a string as argument and displays it in the programming environment. The operator `+` can be applied to two strings, in which case it concatenates them. When it is given two strings, this operator behaves entirely differently than when it is given two numbers. We say that the operator is *polymorphic*. The full language JavaScript allows for more interesting combinations of arguments of `+`, using the concept of *coercion*.

After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for `test_divisor` should not be 2, 3, 4, 5, 6, ... but rather 2, 3, 5, 7, 9, To implement this change, define a function `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest_divisor` function to use `next(test_divisor)` instead of `test_divisor + 1`. With `timed_prime_test` incorporating this modified version of `smallest_divisor`, run the test for each of the 12 primes found in exercise 1.22. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

Exercise 1.24

Modify the `timed_prime_test` function of exercise 1.22 to use `fast_is_prime` (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has $\Theta(\log n)$ growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

Exercise 1.25

Alyssa P. Hacker complains that we went to a lot of extra work in writing `expmod`. After all, she says, since we already know how to compute exponentials, we could have simply written

```
function expmod(base, exp, m) {
    return fast_expt(base, exp) % m;
}
```

Is she correct? Would this function serve as well for our fast prime tester? Explain. [Solution](#)

Exercise 1.26

Louis Reasoner is having great difficulty doing exercise 1.24. His `fast_is_prime` test seems to run more slowly than his `is_prime` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the `expmod` function to use an explicit multiplication, rather than calling `square`:

```

function expmod(base, exp, m) {
    return exp == 0
        ? 1
        : is_even(exp)
            ? expmod(base, exp / 2, m)
                * expmod(base, exp / 2, m)
                % m
            : base
                * expmod(base, exp - 1, m)
                % m;
}

```

'I don't see what difference that could make,' says Louis. 'I do.' says Eva. 'By writing the function like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process.' Explain. [Solution](#)

Exercise 1.27

Demonstrate that the Carmichael numbers listed in footnote [37](#) really do fool the Fermat test. That is, write a function that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your function on the given Carmichael numbers. [Solution](#)

Exercise 1.28

One variant of the Fermat test that cannot be fooled is called the *Miller-Rabin test* (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat's Little Theorem, which states that if n is a prime number and a is any positive integer less than n , then a raised to the $(n - 1)$ st power is congruent to 1 modulo n . To test the primality of a number n by the Miller-Rabin test, we pick a random number $a < n$ and raise a to the $(n - 1)$ st power modulo n using the expmod function. However, whenever we perform the squaring step in expmod, we check to see if we have discovered a 'nontrivial square root of 1 modulo n ', that is, a number not equal to 1 or $n - 1$ whose square is equal to 1 modulo n . It is possible to prove that if such a nontrivial square root of 1 exists, then n is not prime. It is also possible to prove that if n is an odd number that is not prime, then, for at least half the numbers $a < n$, computing a^{n-1} in this way will reveal a nontrivial square root of 1 modulo n . (This is why the Miller-Rabin test cannot be fooled.) Modify the expmod function to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a function analogous to fermat_test. Check your function by testing various known primes and non-primes. Hint: One convenient way to make expmod signal is to have it return 0. [Solution](#)

1.3 Formulating Abstractions with Higher-Order Functions

We have seen that functions are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers. For example, when we declare

```
function cube(x) {
    return x * x * x;
}
```

we are not talking about the cube of a particular number, but rather about a method for obtaining the cube of any number. Of course we could get along without ever declaring this function, by always writing expressions such as

```
3 * 3 * 3;
x * x * x;
y * y * y;
```

and never mentioning `cube` explicitly. This would place us at a serious disadvantage, forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations. Our programs would be able to compute cubes, but our language would lack the ability to express the concept of cubing. One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Functions provide this ability. This is why all but the most primitive programming languages include mechanisms for declaring functions.

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to functions whose parameters must be numbers. Often the same programming pattern will be used with a number of different functions. To express such patterns as concepts, we will need to construct functions that can accept functions as arguments or return functions as values. Functions that manipulate functions are called *higher-order functions*. This section shows how higher-order functions can serve as powerful abstraction mechanisms, vastly increasing the expressive power of our language.

1.3.1 Functions as Arguments

Consider the following three functions. The first computes the sum of the integers from a through b:

```
function sum_integers(a, b) {
    return a > b
        ? 0
        : a + sum_integers(a + 1, b);
}
```

The second computes the sum of the cubes of the integers in the given range:

```
function sum_cubes(a, b) {
    return a > b
        ? 0
        : cube(a) + sum_cubes(a + 1, b);
}
```

The third computes the sum of a sequence of terms in the series

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots$$

which converges to $\pi/8$ (very slowly):⁴⁰

```
function pi_sum(a, b) {
    return a > b
        ? 0
        : 1.0 / (a * (a + 2)) +
            pi_sum(a + 4, b);
}
```

These three functions clearly share a common underlying pattern. They are for the most part identical, differing only in the name of the function, the function of a used to compute the term to be added, and the function that provides the next value of a. We could generate each of the functions by filling in slots in the same template:

```
function name(a, b) {
    return a > b
        ? 0
        : term(a) + name(next(a), b);
}
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Indeed, mathematicians long ago identified the abstraction

⁴⁰This series, usually written in the equivalent form $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$, is due to Leibniz. We'll see how to use this as the basis for some fancy numerical tricks in section 3.5.3.

of *summation of a series* and invented ‘sigma notation,’ for example

$$\sum_{n=a}^b f(n) = f(a) + \cdots + f(b)$$

to express this concept. The power of sigma notation is that it allows mathematicians to deal with the concept of summation itself rather than only with particular sums—for example, to formulate general results about sums that are independent of the particular series being summed.

Similarly, as program designers, we would like our language to be powerful enough so that we can write a function that expresses the concept of summation itself rather than only functions that compute particular sums. We can do so readily in our functional language by taking the common template shown above and transforming the ‘slots’ into parameters:

```
function sum(term, a, next, b) {
    return a > b
        ? 0
        : term(a) + sum(term, next(a), next, b);
}
```

Notice that `sum` takes as its arguments the lower and upper bounds `a` and `b` together with the functions `term` and `next`. We can use `sum` just as we would any function. For example, we can use it (along with a function `inc` that increments its argument by 1) to define `sum_cubes`:

```
function inc(n) {
    return n + 1;
}
function sum_cubes(a, b) {
    return sum(cube, a, inc, b);
}
```

Using this, we can compute the sum of the cubes of the integers from 1 to 10:

```
sum_cubes(1, 10);
```

With the aid of an identity function to compute the term, we can define `sum_integers` in terms of `sum`:

```
function identity(x) {
    return x;
}

function sum_integers(a, b) {
    return sum(identity, a, inc, b);
}
```

Then we can add up the integers from 1 to 10:

```
sum_integers(1, 10);
```

We can also declare `pi_sum` in the same way:⁴¹

```
function pi_sum(a, b) {
    function pi_term(x) {
        return 1.0 / (x * (x + 2));
    }
    function pi_next(x) {
        return x + 4;
    }
    return sum(pi_term, a, pi_next, b);
}
```

Using these functions, we can compute an approximation to π :

```
8 * pi_sum(1, 1000);
```

Once we have `sum`, we can use it as a building block in formulating further concepts. For instance, the definite integral of a function f between the limits a and b can be approximated numerically using the formula

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

for small values of dx . We can express this directly as a function:

```
function integral(f, a, b, dx) {
    function add_dx(x) {
        return x + dx;
    }
    return sum(f, a + dx / 2, add_dx, b) * dx;
}

integral(cube, 0, 1, 0.01);

integral(cube, 0, 1, 0.005);
```

(The exact value of the integral of `cube` between 0 and 1 is 1/4.)

Exercise 1.29

Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function f between a and b is approximated as

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n]$$

⁴¹Notice that we have used block structure (section 1.1.8) to embed the declarations of `pi_next` and `pi_term` within `pi_sum`, since these functions are unlikely to be useful for any other purpose. We will see how to get rid of them altogether in section 1.3.2.

where $h = (b - a)/n$, for some even integer n , and $y_k = f(a + kh)$. (Increasing n increases the accuracy of the approximation.) Declare a function that takes as arguments f , a , b , and n and returns the value of the integral, computed using Simpson's Rule. Use your function to integrate cube between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` function shown above.

[Solution](#)

Exercise 1.30

The `sum` function above generates a linear recursion. The function can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following declaration:

```
function sum(term, a, next, b) {
    function iter(a, result) {
        return ???
        ? ???
        : iter(???, ???);
    }
    return iter(??, ???);
}
```

[Solution](#)

Exercise 1.31

- The `sum` function is only the simplest of a vast number of similar abstractions that can be captured as higher-order functions.⁴² Write an analogous function called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of `product`. Also use `product` to compute approximations to π using the formula⁴³

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

⁴²The intent of exercises 1.31–1.33 is to demonstrate the expressive power that is attained by using an appropriate abstraction to consolidate many seemingly disparate operations. However, though accumulation and filtering are elegant ideas, our hands are somewhat tied in using them at this point since we do not yet have data structures to provide suitable means of combination for these abstractions. We will return to these ideas in section 2.2.3 when we show how to use *sequences* as interfaces for combining filters and accumulators to build even more powerful abstractions. We will see there how these methods really come into their own as a powerful and elegant approach to designing programs.

⁴³This formula was discovered by the seventeenth-century English mathematician John Wallis.

- b. If your product function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

[Solution](#)

Exercise 1.32

- a. Show that sum and product (exercise 1.31) are both special cases of a still more general notion called accumulate that combines a collection of terms, using some general accumulation function:

```
accumulate(combiner, null_value, term, a, next, b);
```

The function accumulate takes as arguments the same term and range specifications as sum and product, together with a combiner function (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a null_value that specifies what base value to use when the terms run out. Write accumulate and show how sum and product can both be declared as simple calls to accumulate.

- b. If your accumulate function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

[Solution](#)

Exercise 1.33

You can obtain an even more general version of accumulate (exercise 1.32) by introducing the notion of a *filter* on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting filtered_accumulate abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write filtered_accumulate as a function. Show how to express the following using filtered_accumulate:

- a. the sum of the squares of the prime numbers in the interval a to b (assuming that you have a `is_prime` predicate already written)

- b. the product of all the positive integers less than n that are relatively prime to n (i.e., all positive integers $i < n$ such that $\text{GCD}(i, n) = 1$).

[Solution](#)

1.3.2 Function Definition Expressions

In using `sum` as in section 1.3.1, it seems terribly awkward to have to declare trivial functions such as `pi_term` and `pi_next` just so we can use them as arguments to our higher-order function. Rather than declare `pi_next` and `pi_term`, it would be more convenient to have a way to directly specify ‘the function that returns its input incremented by 4’ and ‘the function that returns the reciprocal of its input times its input plus 2.’ We can do this by introducing *function definition expressions*, which create functions. Using function definitions, we can describe what we want as

```
x => x + 4;
```

and

```
x => 1.0 / (x * (x + 2));
```

Then our `pi_sum` function can be expressed without declaring any auxiliary functions as

```
function pi_sum(a,b) {
    return sum(x => 1.0 / (x * (x + 2)),
               a,
               x => x + 4,
               b);
}
```

Again using a function definition, we can write the `integral` function without having to declare the auxiliary function `add_dx`:

```
function integral(f, a, b, dx) {
    return sum(f,
               a + dx / 2.0,
               x => x + dx,
               b)
        *
        dx;
}
```

In general, function definitions are used to create functions similarly to function declarations, except that no name is specified for the function⁴⁴ and the `return` keyword along with the

⁴⁴If there is only one parameter, then the parentheses around the parameter list can be omitted.

curly braces can be omitted.⁴⁵

$(\text{parameters}) \Rightarrow \text{expression}$

The resulting function is just as much a function as one that is created using a function declaration statement. The only difference is that it has not been associated with any name in the environment. In fact,

`function plus4(x) { return x + 4; }`

is equivalent to

`const plus4 = x => x + 4;`

We can read a function definition expression as follows:

// read x => x + 4
 // as: ^ ^ ^ ^ ^
 // function with argument x that results in the value plus 4

Like any expression that has a function as its value, a function definition expression can be used as the function expression in an application combination such as

`((x, y, z) => x + y + square(z))(1, 2, 3);`

or, more generally, in any context where we would normally use a function name.⁴⁶ Note that the JavaScript parser requires parentheses around the function in this case.

Using `const` to create local names

Another use of function definitions is in creating local names. We often need local names in our functions other than those that have been bound as parameters. For example, suppose we wish to compute the function

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

⁴⁵In section 2.2.4, we will extend the syntax of function definition expressions to allow blocks as bodies, as in function declaration statements.

⁴⁶The idea to introduce functions without naming them plays a central role in the λ calculus, a mathematical formalism introduced by the mathematical logician Alonzo Church (1941). Church developed the λ calculus to provide a rigorous foundation for studying the notions of function and function application. The λ calculus has become a basic tool for mathematical investigations of the semantics of programming languages.

which we could also express as

$$\begin{aligned} a &= 1 + xy \\ b &= 1 - y \\ f(x, y) &= xa^2 + yb + ab \end{aligned}$$

In writing a function to compute f , we would like to include as local names not only x and y but also the names of intermediate quantities like a and b . One way to accomplish this is to use an auxiliary function to bind the local names:

```
function f(x, y) {
  function f_helper(a, b) {
    return x * square(a) +
           y * b +
           a * b;
  }
  return f_helper(1 + x * y,
                 1 - y);
}
```

Of course, we could use a function definition expression to specify an anonymous function for binding our local names. The body of f then becomes a single call to that function:

```
function f(x,y) {
  return ( (a,b) => x * square(a) +
           y * b +
           a * b
         )(1 + x * y, 1 - y);
}
```

A more convenient way to declare local names is by using **const** within the body of the function. Using **const**, the function f can be written as

```
function f(x, y) {
  const a = 1 + x * y;
  const b = 1 - y;
  return x * square(a) +
           y * b +
           a * b;
}
```

Names that are declared with **const** inside of function declarations have the surrounding block as their scope.⁴⁷

⁴⁷Note that a name declared in a function using **const** cannot be used before the declaration fully is evaluated, not even in the right-hand expression of the declaration itself, and regardless whether the same name is declared outside of the function. Thus the program

Conditional statements

We have seen that it is often useful to declare names that are local to function declarations. When functions become big, it will contribute to their readability if we keep the scope of the names as narrow as possible. Consider for example `expmod` in exercise 1.26 in section 1.2.6.

```
function expmod(base, exp, m) {
    return exp === 0
        ? 1
        : is_even(exp)
            ? expmod(base, exp / 2, m)
                * expmod(base, exp / 2, m)
                % m
            : base
                * expmod(base, exp - 1, m)
                % m;
}
```

This function is unnecessarily inefficient, because it contains two identical calls:

```
expmod(base, exp / 2, m)
```

While this can be easily fixed in this example using the `square` function, this is not so easy in general. Without using `square`, we would be tempted to introduce a local name for the expression as follows:

```
function expmod(base, exp, m) {
    const to_half = expmod(base, exp / 2, m);
    return exp === 0
        ? 1
        : is_even(exp)
            ? to_half * to_half
            % m
        : base
            * expmod(base, exp - 1, m)
            % m;
}
```

This would make the function not just inefficient, but actually non-terminating! The problem is that the constant declaration appears outside the conditional expression, which means that it is executed even when the base case `exp === 0` is reached. To avoid this situation, we shall provide for *conditional statements*, and allow for `return` statements to appear in several branches of the statement. Using a conditional statement, the function `expmod` can be written as follows:

```
function h() {
    const x = 1;
    function i() {
        const x = x + 1;
        return x;
    }
    return i();
}
```

```

function expmod(base, exp, m) {
  if (exp === 0) {
    return 1;
  } else {
    if (is_even(exp)) {
      const to_half = expmod(base, exp / 2, m);
      return to_half * to_half % m;
    } else {
      return base * expmod(base, exp - 1, m) % m;
    }
  }
}

```

The general form of a conditional statement is

```
if (predicate) { consequent } else { alternative }
```

and, like conditional expressions, their evaluation first evaluates the *predicate*. If it evaluates to true, the interpreter evaluates the *consequent* statements and otherwise the *alternative* statements. Note that any constant declarations occurring in either part are local to that part, because both are enclosed in curly braces and thus form their own block.

Exercise 1.34

Suppose we declare

```

function f(g) {
  return g(2);
}

```

Then we have

```

f(square);
f(z => z * (z + 1));

```

What happens if we (perversely) ask the interpreter to evaluate the combination $f(f)$? Explain.

[Solution](#)

1.3.3 Functions as General Methods

We introduced compound functions in section 1.1.4 as a mechanism for abstracting patterns of numerical operations so as to make them independent of the particular numbers involved. With higher-order functions, such as the `integral` function of section 1.3.1, we began to see a more powerful kind of abstraction: functions used to express general methods of computation, independent of the particular functions involved. In this section we discuss two more elaborate examples—general methods for finding zeros and fixed points of functions—and show how these methods can be expressed directly as functions.

Finding roots of equations by the half-interval method

The *half-interval method* is a simple but powerful technique for finding roots of an equation $f(x) = 0$, where f is a continuous function. The idea is that, if we are given points a and b such that $f(a) < 0 < f(b)$, then f must have at least one zero between a and b . To locate a zero, let x be the average of a and b and compute $f(x)$. If $f(x) > 0$, then f must have a zero between a and x . If $f(x) < 0$, then f must have a zero between x and b . Continuing in this way, we can identify smaller and smaller intervals on which f must have a zero. When we reach a point where the interval is small enough, the process stops. Since the interval of uncertainty is reduced by half at each step of the process, the number of steps required grows as $\Theta(\log(L/T))$, where L is the length of the original interval and T is the error tolerance (that is, the size of the interval we will consider ‘small enough’). Here is a function that implements this strategy:⁴⁸

```
function search(f, neg_point, pos_point) {
    const midpoint = average(neg_point, pos_point);
    if (close_enough(neg_point, pos_point)) {
        return midpoint;
    } else {
        const test_value = f(midpoint);
        if (positive(test_value)) {
            return search(f, neg_point, midpoint);
        } else if (negative(test_value)) {
            return search(f, midpoint, pos_point);
        } else {
            return midpoint;
        }
    }
}
```

We assume that we are initially given the function f together with points at which its values are negative and positive. We first compute the midpoint of the two given points. Next we

⁴⁸Note that we slightly extend the syntax of conditional statements described in section 1.3.2 by allowing another conditional statement in place of the block following `else`.

check to see if the given interval is small enough, and if so we simply return the midpoint as our answer. Otherwise, we compute as a test value the value of f at the midpoint. If the test value is positive, then we continue the process with a new interval running from the original negative point to the midpoint. If the test value is negative, we continue with the interval from the midpoint to the positive point. Finally, there is the possibility that the test value is 0, in which case the midpoint is itself the root we are searching for. To test whether the endpoints are ‘close enough’ we can use a function similar to the one used in section 1.1.7 for computing square roots:⁴⁹

```
function close_enough(x, y) {
    return abs(x - y) < 0.001;
}
```

The function search is awkward to use directly, because we can accidentally give it points at which f ’s values do not have the required sign, in which case we get a wrong answer. Instead we will use search via the following function, which checks to see which of the endpoints has a negative function value and which has a positive value, and calls the search function accordingly. If the function has the same sign on the two given points, the half-interval method cannot be used, in which case the function signals an error.⁵⁰

```
function half_interval_method(f, a, b) {
    const a_value = f(a);
    const b_value = f(b);
    return negative(a_value) && positive(b_value)
        ? search(f, a, b)
        : negative(b_value) && positive(a_value)
            ? search(f, b, a)
            : error("values are not of opposite sign");
}
```

The following example uses the half-interval method to approximate π as the root between 2 and 4 of $\sin x = 0$:

```
half_interval_method(math_sin, 2.0, 4.0);
```

Here is another example, using the half-interval method to search for a root of the equation $x^3 - 2x - 3 = 0$ between 1 and 2:

```
half_interval_method(
    x => x * x * x - 2 * x - 3,
    1.0,
    2.0);
```

⁴⁹We have used 0.001 as a representative ‘small’ number to indicate a tolerance for the acceptable error in a calculation. The appropriate tolerance for a real calculation depends upon the problem to be solved and the limitations of the computer and the algorithm. This is often a very subtle consideration, requiring help from a numerical analyst or some other kind of magician.

⁵⁰This can be accomplished using `error`, which takes as argument a string that is printed as error message along with the number of the program line that gave rise to the call of `error`.

Finding fixed points of functions

A number x is called a *fixed point* of a function f if x satisfies the equation $f(x) = x$. For some functions f we can locate a fixed point by beginning with an initial guess and applying f repeatedly,

$$f(x), f(f(x)), f(f(f(x))), \dots$$

until the value does not change very much. Using this idea, we can devise a function `fixed_point` that takes as inputs a function and an initial guess and produces an approximation to a fixed point of the function. We apply the function repeatedly until we find two successive values whose difference is less than some prescribed tolerance:

```
const tolerance = 0.00001;
function fixed_point(f, first_guess) {
    function close_enough(x, y) {
        return abs(x - y) < tolerance;
    }
    function try_with(guess) {
        const next = f(guess);
        return close_enough(guess, next)
            ? next
            : try_with(next);
    }
    return try_with(first_guess);
}
```

For example, we can use this method to approximate the fixed point of the cosine function, starting with 1 as an initial approximation:⁵¹

```
fixed_point(math_cos, 1.0);
```

Similarly, we can find a solution to the equation $y = \sin y + \cos y$:

```
fixed_point(
    y => math_sin(y) + math_cos(y),
    1.0);
```

The fixed-point process is reminiscent of the process we used for finding square roots in section 1.1.7. Both are based on the idea of repeatedly improving a guess until the result satisfies some criterion. In fact, we can readily formulate the square-root computation as a fixed-point search. Computing the square root of some number x requires finding a y such that $y^2 = x$. Putting this equation into the equivalent form $y = x/y$, we recognize that we are looking for a fixed point of the function⁵² $y \mapsto x/y$, and we can therefore try to compute

⁵¹Try this during a boring lecture: Set your calculator to radians mode and then repeatedly press the cos button until you obtain the fixed point.

⁵² \mapsto (pronounced ‘maps to’) is the mathematician’s way of writing function definitions. $y \mapsto x/y$ means $y => x / y$, that is, the function whose value at y is x/y .

square roots as

```
function sqrt(x) {
    return fixed_point(y => x / y, 1.0);
}
// warning: does not converge!
```

Unfortunately, this fixed-point search does not converge. Consider an initial guess y_1 . The next guess is $y_2 = x/y_1$ and the next guess is $y_3 = x/y_2 = x/(x/y_1) = y_1$. This results in an infinite loop in which the two guesses y_1 and y_2 repeat over and over, oscillating about the answer.

One way to control such oscillations is to prevent the guesses from changing so much. Since the answer is always between our guess y and x/y , we can make a new guess that is not as far from y as x/y by averaging y with x/y , so that the next guess after y is $\frac{1}{2}(y + x/y)$ instead of x/y . The process of making such a sequence of guesses is simply the process of looking for a fixed point of $y \mapsto \frac{1}{2}(y + x/y)$:

```
function sqrt(x) {
    return fixed_point(
        y => average(y, x / y),
        1.0);
}
```

(Note that $y = \frac{1}{2}(y + x/y)$ is a simple transformation of the equation $y = x/y$; to derive it, add y to both sides of the equation and divide by 2.)

With this modification, the square-root function works. In fact, if we unravel the definitions, we can see that the sequence of approximations to the square root generated here is precisely the same as the one generated by our original square-root function of section 1.1.7. This approach of averaging successive approximations to a solution, a technique we call *average damping*, often aids the convergence of fixed-point searches.

Exercise 1.35

Show that the golden ratio ϕ (section 1.2.2) is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute ϕ by means of the `fixed_point` function. [Solution](#)

Exercise 1.36

Modify `fixed_point` so that it prints the sequence of approximations it generates, using the primitive function `display` shown in exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log(1000)/\log(x)$. (Use the primitive function `math_log` which computes natural logarithms.) , Compare the number of steps this takes with and without

average damping. (Note that you cannot start `fixed_point` with a guess of 1, as this would cause division by $\log(1) = 0$.)

[Solution](#)

Exercise 1.37

- An infinite *continued fraction* is an expression of the form

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \dots}}}$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\phi$, where ϕ is the golden ratio (described in section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called *k-term finite continued fraction*—has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_K}{D_K}}}$$

Suppose that `n` and `d` are functions of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Declare a function `cont_frac` such that evaluating `cont_frac(n, d, k)` computes the value of the k -term finite continued fraction. Check your function by approximating $1/\phi$ using

```
cont_frac(i => 1.0,
          i => 1.0,
          k);
```

for successive values of k . How large must you make k in order to get an approximation that is accurate to 4 decimal places?

- If your `cont_frac` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

[Solution](#)

Exercise 1.38

In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e - 2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, Write a program that uses your `cont_frac` function from exercise 1.37 to approximate e , based on Euler's expansion.

[Solution](#)

Exercise 1.39

A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \cfrac{x^2}{\ddots}}}}$$

where x is in radians. Declare a function `tan_cf(x, k)` that computes an approximation to the tangent function based on Lambert's formula. As in exercise 1.37, k specifies the number of terms to compute.

[Solution](#)

1.3.4 Functions as Returned Values

The above examples demonstrate how the ability to pass functions as arguments significantly enhances the expressive power of our programming language. We can achieve even more expressive power by creating functions whose returned values are themselves functions.

We can illustrate this idea by looking again at the fixed-point example described at the end of section 1.3.3. We formulated a new version of the square-root function as a fixed-point search, starting with the observation that \sqrt{x} is a fixed-point of the function $y \mapsto x/y$. Then we used average damping to make the approximations converge. Average damping is a useful general technique in itself. Namely, given a function f , we consider the function whose value at x is equal to the average of x and $f(x)$.

We can express the idea of average damping by means of the following function:

```
function average_damp(f) {
    return x => average(x, f(x));
}
```

The function `average_damp` is a function that takes as its argument a function `f` and returns

as its value a function (produced by the function definition expression) that, when applied to a number x , produces the average of x and $f(x)$. For example, applying `average_damp` to the `square` function produces a function whose value at some number x is the average of x and x^2 . Applying this resulting function to 10 returns the average of 10 and 100, or 55:⁵³

```
average_damp(square)(10);
```

Using `average_damp`, we can reformulate the square-root function as follows:

```
function sqrt(x) {
    return fixed_point(average_damp(y => x / y),
                      1.0);
}
```

Notice how this formulation makes explicit the three ideas in the method: fixed-point search, average damping, and the function $y \mapsto x/y$. It is instructive to compare this formulation of the square-root method with the original version given in section 1.1.7. Bear in mind that these functions express the same process, and notice how much clearer the idea becomes when we express the process in terms of these abstractions. In general, there are many ways to formulate a process as a function. Experienced programmers know how to choose process formulations that are particularly perspicuous, and where useful elements of the process are exposed as separate entities that can be reused in other applications. As a simple example of reuse, notice that the cube root of x is a fixed point of the function $y \mapsto x/y^2$, so we can immediately generalize our square-root function to one that extracts cube roots:⁵⁴

```
function cube_root(x) {
    return fixed_point(average_damp(y => x / square(y)),
                      1.0);
}
```

Newton's method

When we first introduced the square-root function, in section 1.1.7, we mentioned that this was a special case of *Newton's method*. If $x \mapsto g(x)$ is a differentiable function, then a solution of the equation $g(x) = 0$ is a fixed point of the function $x \mapsto f(x)$ where

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

and $Dg(x)$ is the derivative of g evaluated at x . Newton's method is the use of the fixed-point method we saw above to approximate a solution of the equation by finding a fixed point of the

⁵³Observe that this is a combination whose operator is itself a combination. Exercise 1.4 already demonstrated the ability to form such combinations, but that was only a toy example. Here we begin to see the real need for such combinations—when applying a function that is obtained as the value returned by a higher-order function.

⁵⁴See exercise 1.45 for a further generalization.

function f .⁵⁵ For many functions g and for sufficiently good initial guesses for x , Newton's method converges very rapidly to a solution of $g(x) = 0$.⁵⁶

In order to implement Newton's method as a function, we must first express the idea of derivative. Note that 'derivative,' like average damping, is something that transforms a function into another function. For instance, the derivative of the function $x \mapsto x^3$ is the function $x \mapsto 3x^2$. In general, if g is a function and dx is a small number, then the derivative Dg of g is the function whose value at any number x is given (in the limit of small dx) by

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

Thus, we can express the idea of derivative (taking dx to be, say, 0.00001) as the function

```
function deriv(g) {
    return x => (g(x + dx) - g(x)) / dx;
}
```

along with the declaration

```
const dx = 0.00001;
```

Like `average_damp`, `deriv` is a function that takes a function as argument and returns a function as value. For example, to approximate the derivative of $x \mapsto x^3$ at 5 (whose exact value is 75) we can evaluate

```
function cube(x) { return x * x * x; }

deriv(cube)(5);
```

With the aid of `deriv`, we can express Newton's method as a fixed-point process:

```
function newton_transform(g) {
    return x => x - g(x) / deriv(g)(x);
}
function newtons_method(g, guess) {
    return fixed_point(newton_transform(g), guess);
}
```

The `newton_transform` function expresses the formula at the beginning of this section, and `newtons_method` is readily defined in terms of this. It takes as arguments a function that computes the function for which we want to find a zero, together with an initial guess. For instance, to find the square root of x , we can use Newton's method to find a zero of the function

⁵⁵Elementary calculus books usually describe Newton's method in terms of the sequence of approximations $x_{n+1} = x_n - g(x_n)/Dg(x_n)$. Having language for talking about processes and using the idea of fixed points simplifies the description of the method.

⁵⁶Newton's method does not always converge to an answer, but it can be shown that in favorable cases each iteration doubles the number-of-digits accuracy of the approximation to the solution. In such cases, Newton's method will converge much more rapidly than the half-interval method.

$y \mapsto y^2 - x$ starting with an initial guess of 1.⁵⁷ This provides yet another form of the square-root function:

```
function sqrt(x) {
    return newtons_method(y => square(y) - x,
                           1.0);
}
```

Abstractions and first-class functions

We've seen two ways to express the square-root computation as an instance of a more general method, once as a fixed-point search and once using Newton's method. Since Newton's method was itself expressed as a fixed-point process, we actually saw two ways to compute square roots as fixed points. Each method begins with a function and finds a fixed point of some transformation of the function. We can express this general idea itself as a function:

```
function fixed_point_of_transform(g, transform, guess) {
    return fixed_point(transform(g), guess);
}
```

This very general function takes as its arguments a function g that computes some function, a function that transforms g , and an initial guess. The returned result is a fixed point of the transformed function.

Using this abstraction, we can recast the first square-root computation from this section (where we look for a fixed point of the average-damped version of $y \mapsto x/y$) as an instance of this general method:

```
function sqrt(x) {
    return fixed_point_of_transform(
        y => x / y,
        average_damp,
        1.0);
}
```

Similarly, we can express the second square-root computation from this section (an instance of Newton's method that finds a fixed point of the Newton transform of $y \mapsto y^2 - x$) as

```
function sqrt(x) {
    return fixed_point_of_transform(
        y => square(y) - x,
        newton_transform,
        1.0);
}
```

We began section 1.3 with the observation that compound functions are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit ele-

⁵⁷For finding square roots, Newton's method converges rapidly to the correct solution from any starting point.

ments in our programming language. Now we've seen how higher-order functions permit us to manipulate these general methods to create further abstractions.

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs and to build upon them and generalize them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order functions is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have *first-class* status. Some of the 'rights and privileges' of first-class elements are:⁵⁸

- They may be referred to using names.
- They may be passed as arguments to functions.
- They may be returned as the results of functions.
- They may be included in data structures.⁵⁹

JavaScript, unlike other common programming languages, awards functions full first-class status. This poses challenges for efficient implementation, but the resulting gain in expressive power is enormous.⁶⁰

Exercise 1.40

Declare a function `cubic` that can be used together with the `newtons_method` function in expressions of the form

```
newtons_method(cubic(a, b, c), 1);
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

[Solution](#)

Exercise 1.41

⁵⁸The notion of first-class status of programming-language elements is due to the British computer scientist Christopher Strachey (1916–1975).

⁵⁹We'll see examples of this after we introduce data structures in chapter 2.

⁶⁰The major implementation cost of first-class functions is that allowing functions to be returned as values requires reserving storage for a function's free names even while the function is not executing. In the JavaScript implementation we will study in section 4.1, these names are stored in the function's environment.

Declare a function `double` that takes a function of one argument as argument and returns a function that applies the original function twice. For example, if `inc` is a function that adds 1 to its argument, then `double(inc)` should be a function that adds 2. What value is returned by

```
double(double(double))(inc)(5); // ?
```

[Solution](#)

Exercise 1.42

Let f and g be two one-argument functions. The *composition* f after g is defined to be the function $x \mapsto f(g(x))$. Declare a function `compose` that implements composition. For example, if `inc` is a function that adds 1 to its argument,

```
compose(square, inc)(6);
returns 49.
```

[Solution](#)

Exercise 1.43

If f is a numerical function and n is a positive integer, then we can form the n th repeated application of f , which is defined to be the function whose value at x is $f(f(\dots(f(x))\dots))$. For example, if f is the function $x \mapsto x + 1$, then the n th repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the n th repeated application of f is the function that raises its argument to the 2^n th power. Write a function that takes as inputs a function that computes f and a positive integer n and returns the function that computes the n th repeated application of f . Your function should be able to be used as follows:

```
repeated(square, 2)(5);
```

Hint: You may find it convenient to use `compose` from exercise 1.42.

[Solution](#)

Exercise 1.44

The idea of *smoothing* a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x - dx)$, $f(x)$, and $f(x + dx)$. Write a function `smooth` that takes as input a function that computes f and returns a function that computes the smoothed f . It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the n -fold smoothed function. Show how to generate the n -fold smoothed

function of any given function using smooth and repeated from exercise 1.43. [Solution](#)

Exercise 1.45

We saw in section 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute n th roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^{n-1}$. Use this to implement a simple function for computing n th roots using `fixed_point`, `average_damp`, and the repeated function of exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

[Solution](#)

Exercise 1.46

Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as *iterative improvement*. Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a function `iterative_improve` that takes two functions as arguments: a method for telling whether a guess is good enough and a method for improving a guess. The function `iterative_improve` should return as its value a function that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` function of section 1.1.7 and the `fixed_point` function of section 1.3.3 in terms of `iterative_improve`.

[Solution](#)

Chapter 2

Building Abstractions with Data

We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. . . . [The mathematician] need not be idle; there are many operations which he may carry out with these symbols, without ever having to look at the things they stand for.

— Hermann Weyl, *The Mathematical Way of Thinking*

We concentrated in chapter 1 on computational processes and on the role of functions in program design. We saw how to use primitive data (numbers) and primitive operations (arithmetic operations), how to combine functions to form compound functions through composition, conditionals, and the use of parameters, and how to abstract processes by using function declarations. We saw that a function can be regarded as a pattern for the local evolution of a process, and we classified, reasoned about, and performed simple algorithmic analyses of some common patterns for processes as embodied in functions. We also saw that higher-order functions enhance the power of our language by enabling us to manipulate, and thereby to reason in terms of, general methods of computation. This is much of the essence of programming.

In this chapter we are going to look at more complex data. All the functions in chapter 1 operate on simple numerical data, and simple data are not sufficient for many of the problems we wish to address using computation. Programs are typically designed to model complex phenomena, and more often than not one must construct computational objects that have several parts in order to model real-world phenomena that have several aspects. Thus, whereas our focus in chapter 1 was on building abstractions by combining functions to form compound functions, we turn in this chapter to another key aspect of any programming language: the means it provides for building abstractions by combining data objects to form *compound data*.

Why do we want compound data in a programming language? For the same reasons that

we want compound functions: to elevate the conceptual level at which we can design our programs, to increase the modularity of our designs, and to enhance the expressive power of our language. Just as the ability to declare functions enables us to deal with processes at a higher conceptual level than that of the primitive operations of the language, the ability to construct compound data objects enables us to deal with data at a higher conceptual level than that of the primitive data objects of the language.

Consider the task of designing a system to perform arithmetic with rational numbers. We could imagine an operation `add_rat` that takes two rational numbers and produces their sum. In terms of simple data, a rational number can be thought of as two integers: a numerator and a denominator. Thus, we could design a program in which each rational number would be represented by two integers (a numerator and a denominator) and where `add_rat` would be implemented by two functions (one producing the numerator of the sum and one producing the denominator). But this would be awkward, because we would then need to explicitly keep track of which numerators corresponded to which denominators. In a system intended to perform many operations on many rational numbers, such bookkeeping details would clutter the programs substantially, to say nothing of what they would do to our minds. It would be much better if we could ‘glue together’ a numerator and denominator to form a pair—a *compound data object*—that our programs could manipulate in a way that would be consistent with regarding a rational number as a single conceptual unit.

The use of compound data also enables us to increase the modularity of our programs. If we can manipulate rational numbers directly as objects in their own right, then we can separate the part of our program that deals with rational numbers per se from the details of how rational numbers may be represented as pairs of integers. The general technique of isolating the parts of a program that deal with how data objects are represented from the parts of a program that deal with how data objects are used is a powerful design methodology called *data abstraction*. We will see how data abstraction makes programs much easier to design, maintain, and modify.

The use of compound data leads to a real increase in the expressive power of our programming language. Consider the idea of forming a ‘linear combination’ $ax + by$. We might like to write a function that would accept a , b , x , and y as arguments and return the value of $ax + by$. This presents no difficulty if the arguments are to be numbers, because we can readily declare the function

```
function linear_combination(a, b, x, y) {
    return a * x + b * y;
}
```

But suppose we are not concerned only with numbers. Suppose we would like to describe a process that forms linear combinations whenever addition and multiplication are defined—for rational numbers, complex numbers, polynomials, or whatever. We could express this as a function of the form

```
function linear_combination(a, b, x, y) {
    return add(mul(a, x), mul(b, y));
}
```

where `add` and `mul` are not the primitive functions `+` and `*` but rather more complex things that will perform the appropriate operations for whatever kinds of data we pass in as the arguments `a`, `b`, `x`, and `y`. The key point is that the only thing `linear_combination` should need to know about `a`, `b`, `x`, and `y` is that the functions `add` and `mul` will perform the appropriate manipulations. From the perspective of the function `linear_combination`, it is irrelevant what `a`, `b`, `x`, and `y` are and even more irrelevant how they might happen to be represented in terms of more primitive data. This same example shows why it is important that our programming language provide the ability to manipulate compound objects directly: Without this, there is no way for a function such as `linear_combination` to pass its arguments along to `add` and `mul` without having to know their detailed structure.¹

We begin this chapter by implementing the rational-number arithmetic system mentioned above. This will form the background for our discussion of compound data and data abstraction. As with compound functions, the main issue to be addressed is that of abstraction as a technique for coping with complexity, and we will see how data abstraction enables us to erect suitable *abstraction barriers* between different parts of a program.

We will see that the key to forming compound data is that a programming language should provide some kind of ‘glue’ so that data objects can be combined to form more complex data objects. There are many possible kinds of glue. Indeed, we will discover how to form compound data using no special ‘data’ operations at all, only functions. This will further blur the distinction between ‘function’ and ‘data,’ which was already becoming tenuous toward the end of chapter 1. We will also explore some conventional techniques for representing sequences and trees. One key idea in dealing with compound data is the notion of *closure*—that the glue we use for combining data objects should allow us to combine not only primitive data objects, but compound data objects as well. Another key idea is that compound data objects can serve as *conventional interfaces* for combining program modules in mix-and-match ways. We illustrate some of these ideas by presenting a simple graphics language that exploits closure.

We will then augment the representational power of our language by introducing *symbolic expressions*—data whose elementary parts can be arbitrary symbols rather than only numbers. We explore various alternatives for representing sets of objects. We will find that, just as a given

¹The ability to directly manipulate functions provides an analogous increase in the expressive power of a programming language. For example, in section 1.3.1 we introduced the `sum` function, which takes a function `term` as an argument and computes the sum of the values of `term` over some specified interval. In order to define `sum`, it is crucial that we be able to speak of a function such as `term` as an entity in its own right, without regard for how `term` might be expressed with more primitive operations. Indeed, if we did not have the notion of ‘a function,’ it is doubtful that we would ever even think of the possibility of defining an operation such as `sum`. Moreover, insofar as performing the summation is concerned, the details of how `term` may be constructed from more primitive operations are irrelevant.

numerical function can be computed by many different computational processes, there are many ways in which a given data structure can be represented in terms of simpler objects, and the choice of representation can have significant impact on the time and space requirements of processes that manipulate the data. We will investigate these ideas in the context of symbolic differentiation, the representation of sets, and the encoding of information.

Next we will take up the problem of working with data that may be represented differently by different parts of a program. This leads to the need to implement *generic operations*, which must handle many different types of data. Maintaining modularity in the presence of generic operations requires more powerful abstraction barriers than can be erected with simple data abstraction alone. In particular, we introduce *data-directed programming* as a technique that allows individual data representations to be designed in isolation and then combined *additively* (i.e., without modification). To illustrate the power of this approach to system design, we close the chapter by applying what we have learned to the implementation of a package for performing symbolic arithmetic on polynomials, in which the coefficients of the polynomials can be integers, rational numbers, complex numbers, and even other polynomials.

2.1 Introduction to Data Abstraction

In section 1.1.8, we noted that a function used as an element in creating a more complex function could be regarded not only as a collection of particular operations but also as a functional abstraction. That is, the details of how the function was implemented could be suppressed, and the particular function itself could be replaced by any other function with the same overall behavior. In other words, we could make an abstraction that would separate the way the function would be used from the details of how the function would be implemented in terms of more primitive functions. The analogous notion for compound data is called *data abstraction*. Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects.

The basic idea of data abstraction is to structure the programs that are to use compound data objects so that they operate on ‘abstract data.’ That is, our programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand. At the same time, a ‘concrete’ data representation is defined independent of the programs that use the data. The interface between these two parts of our system will be a set of functions, called *selectors* and *constructors*, that implement the abstract data in terms of the concrete representation. To illustrate this technique, we will consider how to design a set of functions for manipulating rational numbers.

2.1.1 Example: Arithmetic Operations for Rational Numbers

Suppose we want to do arithmetic with rational numbers. We want to be able to add, subtract, multiply, and divide them and to test whether two rational numbers are equal.

Let us begin by assuming that we already have a way of constructing a rational number from a numerator and a denominator. We also assume that, given a rational number, we have a way of extracting (or selecting) its numerator and its denominator. Let us further assume that the constructor and selectors are available as functions:

- `make_rat(n, d)` returns the rational number whose numerator is the integer `n` and whose denominator is the integer `d`.
- `numer(x)` returns the numerator of the rational number `x`.
- `denom(x)` returns the denominator of the rational number `x`.

We are using here a powerful strategy of synthesis: *wishful thinking*. We haven't yet said how a rational number is represented, or how the functions `numer`, `denom`, and `make_rat` should be implemented. Even so, if we did have these three functions, we could then add, subtract, multiply, divide, and test equality by using the following relations:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1d_2 + n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1d_2 - n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1n_2}{d_1d_2}$$

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1d_2}{d_1n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \text{ if and only if } n_1d_2 = n_2d_1$$

We can express these rules as functions:

```
function add_rat(x, y) {
    return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                    denom(x) * denom(y));
}

function sub_rat(x, y) {
    return make_rat(numer(x) * denom(y) - numer(y) * denom(x),
                    denom(x) * denom(y));
}
```

```

        denom(x) * denom(y));
}
function mul_rat(x, y) {
    return make_rat(numer(x) * numer(y),
                    denom(x) * denom(y));
}
function div_rat(x, y) {
    return make_rat(numer(x) * denom(y),
                    denom(x) * numer(y));
}
function equal_rat(x, y) {
    return numer(x) * denom(y) === numer(y) * denom(x);
}

```

Now we have the operations on rational numbers defined in terms of the selector and constructor functions `numer`, `denom`, and `make_rat`. But we haven't yet defined these. What we need is some way to glue together a numerator and a denominator to form a rational number.

Pairs

To enable us to implement the concrete level of our data abstraction, our language provides a compound structure called a *pair*, which can be constructed with the function `pair`. This function takes two arguments and returns a compound data object that contains the two arguments as parts. Given a pair, we can extract the parts using the functions `head` and `tail`. These functions are not 'primitive' functions in JavaScript, but the programs in this book treat them as if they were.² Thus, we can use `pair`, `head`, and `tail` as follows:

```

const x = pair(1,2);

head(x);

tail(x);

```

Notice that a pair is a data object that can be given a name and manipulated, just like a primitive data object. Moreover, `pair` can be used to form pairs whose elements are pairs, and so on:

```

const x = pair(1,2);
const y = pair(3,4);
const z = pair(x,y);

head(head(z));

head(tail(z));

```

In section 2.2 we will see how this ability to combine pairs means that pairs can be used

²In practice, we provide these functions in a *library* and make sure that the library is loaded whenever we need them. One way of defining these functions is to use *arrays*, one of JavaScript's primitive data structures.

as general-purpose building blocks to create all sorts of complex data structures. The single compound-data primitive *pair*, implemented by the functions `pair`, `head`, and `tail`, is the only glue we need. Data objects constructed from pairs are called *list-structured* data.

Representing rational numbers

Pairs offer a natural way to complete the rational-number system. Simply represent a rational number as a pair of two integers: a numerator and a denominator. Then `make_rat`, `numer`, and `denom` are readily implemented as follows:³

```
function make_rat(n, d) {
    return pair(n, d);
}
function numer(x) {
    return head(x);
}
function denom(x) {
    return tail(x);
}
```

Also, in order to display the results of our computations, we can print rational numbers by printing the numerator, a slash, and the denominator:⁴

```
function print_rat(x) {
    display(numer(x));
    display("-");
    display(denom(x));
}
```

Now we can try our rational-number functions:

```
const one_half = make_rat(1, 2);

print_rat(one_half);
```

³Another way to define the selectors and constructor is

```
const make_rat = pair;
const numer = head;
const denom = tail;
```

The first definition associates the name `make_rat` with the value of the expression `pair`, which is the primitive function that constructs pairs. Thus `make_rat` and `pair` are names for the same primitive constructor.

Defining selectors and constructors in this way is efficient: Instead of `make_rat` calling `pair`, `make_rat` is `pair`, so there is only one function called, not two, when `make_rat` is called. On the other hand, doing this defeats debugging aids that trace function calls or put breakpoints on function calls: You may want to watch `make_rat` being called, but you certainly don't want to watch every call to `pair`.

We have chosen not to use this style of definition in this book.

⁴The primitive function `display` is our facility for printing data. We display a rational number by printing its denominator in one line, a dash in the next line and its numerator in the last line.

```

const one_third = make_rat(1, 3);

print_rat(one_third);

print_rat(add_rat(one_half, one_third));

print_rat(mul_rat(one_half, one_third));

print_rat(div_rat(one_half, one_third));

```

As the final example shows, our rational-number implementation does not reduce rational numbers to lowest terms. We can remedy this by changing `make_rat`. If we have a `gcd` function like the one in section 1.2.5 that produces the greatest common divisor of two integers, we can use `gcd` to reduce the numerator and the denominator to lowest terms before constructing the pair:

```

function make_rat(n, d) {
  const g = gcd(n, d);
  return pair(n / g, d / g);
}

```

Now we have

```

function numer(x) {
  return head(x);
}
function denom(x) {
  return tail(x);
}
function add_rat(x, y) {
  return make_rat(numer(x) * denom(y) + numer(y) * denom(x),
                  denom(x) * denom(y));
}
function sub_rat(x, y) {
  return make_rat(numer(x) * denom(y) - numer(y) * denom(x),
                  denom(x) * denom(y));
}
function mul_rat(x, y) {
  return make_rat(numer(x) * numer(y),
                  denom(x) * denom(y));
}
function div_rat(x, y) {
  return make_rat(numer(x) * denom(y),
                  denom(x) * numer(y));
}
function equal_rat(x, y) {
  return numer(x) * denom(y) === numer(y) * denom(x);
}

print_rat(add_rat(one_third, one_third));

```

as desired. This modification was accomplished by changing the constructor `make_rat` without changing any of the functions (such as `add_rat` and `mul_rat`) that implement the actual operations.

Exercise 2.1

Define a better version of `make_rat` that handles both positive and negative arguments. The function `make_rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

[Solution](#)

2.1.2 Abstraction Barriers

Before continuing with more examples of compound data and data abstraction, let us consider some of the issues raised by the rational-number example. We defined the rational-number operations in terms of a constructor `make_rat` and selectors `numer` and `denom`. In general, the underlying idea of data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, and then to use only those operations in manipulating the data.

We can envision the structure of the rational-number system as shown in Figure 2.1. The horizontal lines represent *abstraction barriers* that isolate different ‘levels’ of the system. At each level, the barrier separates the programs (above) that use the data abstraction from the programs (below) that implement the data abstraction. Programs that use rational numbers manipulate them solely in terms of the functions supplied ‘for public use’ by the rational-number package: `add_rat`, `sub_rat`, `mul_rat`, `div_rat`, and `equal_rat`. These, in turn, are implemented solely in terms of the constructor and selectors `make_rat`, `numer`, and `denom`, which themselves are implemented in terms of pairs. The details of how pairs are implemented are irrelevant to the rest of the rational-number package so long as pairs can be manipulated by the use of `pair`, `head`, and `tail`. In effect, functions at each level are the interfaces that define the abstraction barriers and connect the different levels.

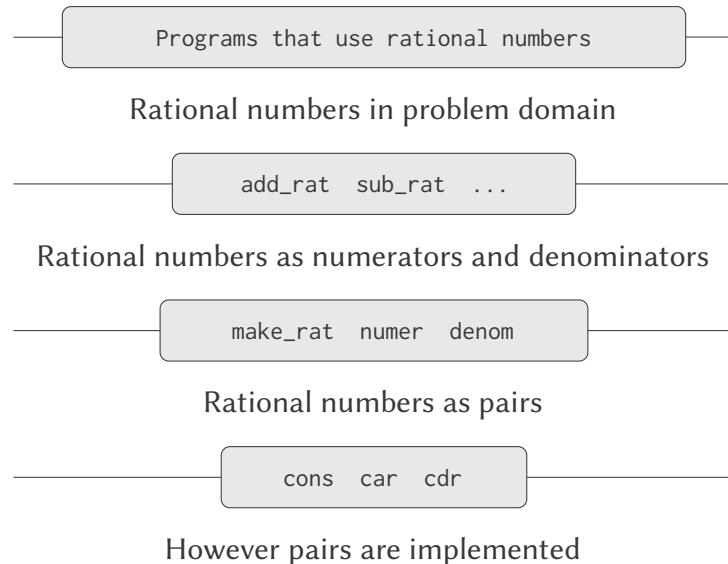


Figure 2.1: Data-abstraction barriers in the rational-number package.

This simple idea has many advantages. One advantage is that it makes programs much easier to maintain and to modify. Any complex data structure can be represented in a variety of ways with the primitive data structures provided by a programming language. Of course, the choice of representation influences the programs that operate on it; thus, if the representation were to be changed at some later time, all such programs might have to be modified accordingly. This task could be time-consuming and expensive in the case of large programs unless the dependence on the representation were to be confined by design to a very few program modules.

For example, an alternate way to address the problem of reducing rational numbers to lowest terms is to perform the reduction whenever we access the parts of a rational number, rather than when we construct it. This leads to different constructor and selector functions:

```

function make_rat(n, d) {
    return pair(n, d);
}
function numer(x) {
    const g = gcd(head(x), tail(x));
    return head(x) / g;
}
function denom(x) {
    const g = gcd(head(x), tail(x));
    return tail(x) / g;
}
  
```

The difference between this implementation and the previous one lies in when we compute the gcd. If in our typical use of rational numbers we access the numerators and denominators of the same rational numbers many times, it would be preferable to compute the gcd when

the rational numbers are constructed. If not, we may be better off waiting until access time to compute the gcd. In any case, when we change from one representation to the other, the functions `add_rat`, `sub_rat`, and so on do not have to be modified at all.

Constraining the dependence on the representation to a few interface functions helps us design programs as well as modify them, because it allows us to maintain the flexibility to consider alternate implementations. To continue with our simple example, suppose we are designing a rational-number package and we can't decide initially whether to perform the gcd at construction time or at selection time. The data-abstraction methodology gives us a way to defer that decision without losing the ability to make progress on the rest of the system.

Exercise 2.2

Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Declare a constructor `make_segment` and selectors `start_segment` and `end_segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the *x* coordinate and the *y* coordinate. Accordingly, specify a constructor `make_point` and selectors `x_point` and `y_point` that define this representation. Finally, using your selectors and constructors, declare a function `midpoint_segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your functions, you'll need a way to print points:

```
function print_point(p) {
    display("(");
    display(x_point(p));
    display(",");
    display(y_point(p));
    display(")");
}
```

[Solution](#)

Exercise 2.3

Implement a representation for rectangles in a plane. (Hint: You may want to make use of exercise 2.2.) In terms of your constructors and selectors, create functions that compute the perimeter and the area of a given rectangle. Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area functions will work using either representation? [Solution](#)

2.1.3 What Is Meant by Data?

We began the rational-number implementation in section 2.1.1 by implementing the rational-number operations `add_rat`, `sub_rat`, and so on in terms of three unspecified functions: `make_rat`, `numer`, and `denom`. At that point, we could think of the operations as being defined in terms of data objects—numerators, denominators, and rational numbers—whose behavior was specified by the latter three functions.

But exactly what is meant by *data*? It is not enough to say ‘whatever is implemented by the given selectors and constructors.’ Clearly, not every arbitrary set of three functions can serve as an appropriate basis for the rational-number implementation. We need to guarantee that, if we construct a rational number x from a pair of integers n and d , then extracting the `numer` and the `denom` of x and dividing them should yield the same result as dividing n by d . In other words, `make_rat`, `numer`, and `denom` must satisfy the condition that, for any integer n and any non-zero integer d , if x is `make_rat(n, d)`, then

$$\frac{\text{numer}(x)}{\text{denom}(x)} = \frac{n}{d}$$

In fact, this is the only condition `make_rat`, `numer`, and `denom` must fulfill in order to form a suitable basis for a rational-number representation. In general, we can think of data as defined by some collection of selectors and constructors, together with specified conditions that these functions must fulfill in order to be a valid representation.⁵

This point of view can serve to define not only ‘high-level’ data objects, such as rational numbers, but lower-level objects as well. Consider the notion of a pair, which we used in order to define our rational numbers. We never actually said what a pair was, only that the language supplied functions `pair`, `head`, and `tail` for operating on pairs. But the only thing we need to know about these three operations is that if we glue two objects together using `pair` we can retrieve the objects using `head` and `tail`. That is, the operations satisfy the condition that, for any objects x and y , if z is `pair(x, y)` then `head(z)` is x and `tail(z)` is y . Indeed, we mentioned that these three functions are included as primitives in our language. However, any triple of functions that satisfies the above condition can be used as the basis for implementing pairs.

⁵Surprisingly, this idea is very difficult to formulate rigorously. There are two approaches to giving such a formulation. One, pioneered by C. A. R. Hoare (1972), is known as the method of *abstract models*. It formalizes the ‘functions plus conditions’ specification as outlined in the rational-number example above. Note that the condition on the rational-number representation was stated in terms of facts about integers (equality and division). In general, abstract models define new kinds of data objects in terms of previously defined types of data objects. Assertions about data objects can therefore be checked by reducing them to assertions about previously defined data objects. Another approach, introduced by Zilles at MIT, by Goguen, Thatcher, Wagner, and Wright at IBM (see Thatcher, Wagner, and Wright 1978), and by Guttag at Toronto (see Guttag 1977), is called *algebraic specification*. It regards the ‘functions’ as elements of an abstract algebraic system whose behavior is specified by axioms that correspond to our ‘conditions,’ and uses the techniques of abstract algebra to check assertions about data objects. Both methods are surveyed in the paper by Liskov and Zilles (1975).

This point is illustrated strikingly by the fact that we could implement pair, head, and tail without using any data structures at all but only using functions. Here are the definitions: The function `Error` is similar to the function `error` but takes a second argument, which is converted to a string and then concatenated with the first argument.

```
function pair(x, y) {
  return m =>
    m === 0
    ? x
    : m === 1
    ? y
    : Error("Argument not 0 or 1 in pair", m);
}
function head(z) {
  return z(0);
}
function tail(z) {
  return z(1);
}
```

This use of functions corresponds to nothing like our intuitive notion of what data should be. Nevertheless, all we need to do to show that this is a valid way to represent pairs is to verify that these functions satisfy the condition given above.

The subtle point to notice is that the value returned by `pair(x, y)` is a function—namely the internally defined function `dispatch`, which takes one argument and returns either `x` or `y` depending on whether the argument is 0 or 1. Correspondingly, `head(z)` is defined to apply `z` to 0. Hence, if `z` is the function formed by `pair(x, y)`, then `z` applied to 0 will yield `x`. Thus, we have shown that `head(pair(x, y))` yields `x`, as desired. Similarly, `tail(pair(x, y))` applies the function returned by `pair(x, y)` to 1, which returns `y`. Therefore, this functional implementation of pairs is a valid implementation, and if we access pairs using only `pair`, `head`, and `tail` we cannot distinguish this implementation from one that uses ‘real’ data structures.

The point of exhibiting the functional representation of pairs is not that our language works this way (we will be using arrays to represent pairs) but that it could work this way. The functional representation, although obscure, is a perfectly adequate way to represent pairs, since it fulfills the only conditions that pairs need to fulfill. This example also demonstrates that the ability to manipulate functions as objects automatically provides the ability to represent compound data. This may seem a curiosity now, but functional representations of data will play a central role in our programming repertoire. This style of programming is often called *message passing*, and we will be using it as a basic tool in chapter 3 when we address the issues of modeling and simulation.

Exercise 2.4

Here is an alternative functional representation of pairs. For this representation, verify that `head(pair(x, y))` yields `x` for any objects `x` and `y`.

```
function pair(x, y) {
    return m => m(x, y);
}
function head(z) {
    return z((p, q) => p);
}
```

What is the corresponding definition of `tail`? (Hint: To verify that this works, make use of the substitution model of section 1.1.5.)

[Solution](#)

Exercise 2.5

Show that we can represent pairs of nonnegative integers using only numbers and arithmetic operations if we represent the pair a and b as the integer that is the product $2^a 3^b$. Give the corresponding definitions of the functions `pair`, `head`, and `tail`.

[Solution](#)

Exercise 2.6

In case representing pairs as functions wasn't mind-boggling enough, consider that, in a language that can manipulate functions, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
const zero = f => x => x;
function add_1(n) {
    return f => x => f(n(f))(x));
}
```

This representation is known as *Church numerals*, after its inventor, Alonzo Church, the logician who invented the λ calculus. Define one and two directly (not in terms of zero and `add_1`). (Hint: Use substitution to evaluate `add_1(zero)`). Give a direct definition of the addition function `plus` (not in terms of repeated application of `add_1`).

[Solution](#)

2.1.4 Extended Exercise: Interval Arithmetic

Alyssa P. Hacker is designing a system to help people solve engineering problems. One feature she wants to provide in her system is the ability to manipulate inexact quantities (such as measured parameters of physical devices) with known precision, so that when computations are done with such approximate quantities the results will be numbers of known precision.

Electrical engineers will be using Alyssa's system to compute electrical quantities. It is sometimes necessary for them to compute the value of a parallel equivalent resistance R_p of two resistors R_1 and R_2 using the formula

$$R_p = \frac{1}{1/R_1 + 1/R_2}$$

Resistance values are usually known only up to some tolerance guaranteed by the manufacturer of the resistor. For example, if you buy a resistor labeled '6.8 ohms with 10% tolerance' you can only be sure that the resistor has a resistance between $6.8 - 0.68 = 6.12$ and $6.8 + 0.68 = 7.48$ ohms. Thus, if you have a 6.8-ohm 10% resistor in parallel with a 4.7-ohm 5% resistor, the resistance of the combination can range from about 2.58 ohms (if the two resistors are at the lower bounds) to about 2.97 ohms (if the two resistors are at the upper bounds).

Alyssa's idea is to implement 'interval arithmetic' as a set of arithmetic operations for combining 'intervals' (objects that represent the range of possible values of an inexact quantity). The result of adding, subtracting, multiplying, or dividing two intervals is itself an interval, representing the range of the result.

Alyssa postulates the existence of an abstract object called an 'interval' that has two endpoints: a lower bound and an upper bound. She also presumes that, given the endpoints of an interval, she can construct the interval using the data constructor `make_interval`. Alyssa first writes a function for adding two intervals. She reasons that the minimum value the sum could be is the sum of the two lower bounds and the maximum value it could be is the sum of the two upper bounds:

```
function add_interval(x, y) {
    return make_interval(lower_bound(x) + lower_bound(y),
                         upper_bound(x) + upper_bound(y));
}
```

Alyssa also works out the product of two intervals by finding the minimum and the maximum of the products of the bounds and using them as the bounds of the resulting interval. (`math_min` and `math_max` are primitives that find the minimum or maximum of any number of arguments.)

```
function mul_interval(x, y) {
  const p1 = lower_bound(x) * lower_bound(y);
  const p2 = lower_bound(x) * upper_bound(y);
  const p3 = upper_bound(x) * lower_bound(y);
  const p4 = upper_bound(x) * upper_bound(y);
  return make_interval(math_min(p1, p2, p3, p4),
                      math_max(p1, p2, p3, p4));
}
```

To divide two intervals, Alyssa multiplies the first by the reciprocal of the second. Note that the bounds of the reciprocal interval are the reciprocal of the upper bound and the reciprocal of the lower bound, in that order.

```
function div_interval(x,y) {
  return mul_interval(x, make_interval(1 / upper_bound(y),
                                      1 / lower_bound(y)));
}
```

Exercise 2.7

Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Here is a definition of the interval constructor:

```
function make_interval(x, y) {
  return pair(x, y);
}
// lower_bound ...
// upper_bound ...
```

Define selectors `upper_bound` and `lower_bound` to complete the implementation. [Solution](#)

Exercise 2.8

Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction function, called `sub_interval`. [Solution](#)

Exercise 2.9

The *width* of an interval is half of the difference between its upper and lower bounds. The width is a measure of the uncertainty of the number specified by the interval. For some arithmetic operations the width of the result of combining two intervals is a function only of the widths of the argument intervals, whereas for others the width of the combination is not a function

of the widths of the argument intervals. Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted). Give examples to show that this is not true for multiplication or division. [Solution](#)

Exercise 2.10

Ben Bitdiddle, an expert systems programmer, looks over Alyssa's shoulder and comments that it is not clear what it means to divide by an interval that spans zero. Modify Alyssa's program to check for this condition and to signal an error if it occurs. [Solution](#)

Exercise 2.11

In passing, Ben also cryptically comments: 'By testing the signs of the endpoints of the intervals, it is possible to break `mul_interval` into nine cases, only one of which requires more than two multiplications.' Rewrite this function using Ben's suggestion. [Solution](#)

After debugging her program, Alyssa shows it to a potential user, who complains that her program solves the wrong problem. He wants a program that can deal with numbers represented as a center value and an additive tolerance; for example, he wants to work with intervals such as 3.5 ± 0.15 rather than $[3.35, 3.65]$. Alyssa returns to her desk and fixes this problem by supplying an alternate constructor and alternate selectors:

```
function make_center_width(c, w) {
    return make_interval(c - w, c + w);
}
function center(i) {
    return (lower_bound(i) + upper_bound(i)) / 2;
}
function width(i) {
    return (upper_bound(i) - lower_bound(i)) / 2;
}
```

Unfortunately, most of Alyssa's users are engineers. Real engineering situations usually involve measurements with only a small uncertainty, measured as the ratio of the width of the interval to the midpoint of the interval. Engineers usually specify percentage tolerances on the parameters of devices, as in the resistor specifications given earlier.

Exercise 2.12

Define a constructor `make_center_percent` that takes a center and a percentage tolerance

and produces the desired interval. You must also define a selector `percent` that produces the percentage tolerance for a given interval. The center selector is the same as the one shown above.

[Solution](#)

Exercise 2.13

Show that under the assumption of small percentage tolerances there is a simple formula for the approximate percentage tolerance of the product of two intervals in terms of the tolerances of the factors. You may simplify the problem by assuming that all numbers are positive. [Solution](#)

After considerable work, Alyssa P. Hacker delivers her finished system. Several years later, after she has forgotten all about it, she gets a frenzied call from an irate user, Lem E. Tweakit. It seems that Lem has noticed that the formula for parallel resistors can be written in two algebraically equivalent ways:

$$\frac{R_1 R_2}{R_1 + R_2}$$

and

$$\frac{1}{1/R_1 + 1/R_2}$$

He has written the following two programs, each of which computes the parallel-resistors formula differently:

```
function par1(r1, r2) {
    return div_interval(mul_interval(r1, r2),
                        add_interval(r1, r2));
}
function par2(r1, r2) {
    const one = make_interval(1, 1);
    return div_interval(one,
                        add_interval(div_interval(one, r1),
                                    div_interval(one, r2)));
}
```

Lem complains that Alyssa's program gives different answers for the two ways of computing. This is a serious complaint.

Exercise 2.14

Demonstrate that Lem is right. Investigate the behavior of the system on a variety of arithmetic expressions. Make some intervals A and B , and use them in computing the expressions A/A and A/B . You will get the most insight by using intervals whose width is a small percentage of the center value. Examine the results of the computation in center-percent form (see exer-

cise 2.12).

[Solution](#)

Exercise 2.15

Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that a formula to compute with intervals using Alyssa’s system will produce tighter error bounds if it can be written in such a form that no name that represents an uncertain number is repeated. Thus, she says, `par2` is a ‘better’ program for parallel resistances than `par1`. Is she right? Why?

[Solution](#)

Exercise 2.16

Explain, in general, why equivalent algebraic expressions may lead to different answers. Can you devise an interval-arithmetic package that does not have this shortcoming, or is this task impossible? (Warning: This problem is very difficult.)

[Solution](#)

2.2 Hierarchical Data and the Closure Property

As we have seen, pairs provide a primitive ‘glue’ that we can use to construct compound data objects. Figure 2.2 shows a standard way to visualize a pair—in this case, the pair formed by `pair(1, 2)`.

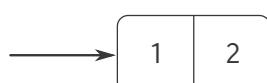


Figure 2.2: Box-and-pointer representation of `pair(1, 2)`.

In this representation, which is called *box-and-pointer notation*, each compound object is shown as a *pointer* to a box. The box for a pair has two parts, the left part containing the head of the pair and the right part containing the tail.⁶

We have already seen that `pair` can be used to combine not only numbers but pairs as well. (You made use of this fact, or should have, in doing exercises 2.2 and 2.3.) As a consequence, pairs provide a universal building block from which we can construct all sorts of data structures. Figure 2.3 shows two ways to use pairs to combine the numbers 1, 2, 3, and 4.

⁶In this JavaScript adaptation, we choose to draw primitive objects directly inside of the boxes of the pairs that contain them, in an attempt to be consistent with a similar practice introduced in section 3.2. The box-and-pointer diagrams in the original version of the textbook include separate boxes for primitive objects, such as 1 and 2, each containing a representation of the object.

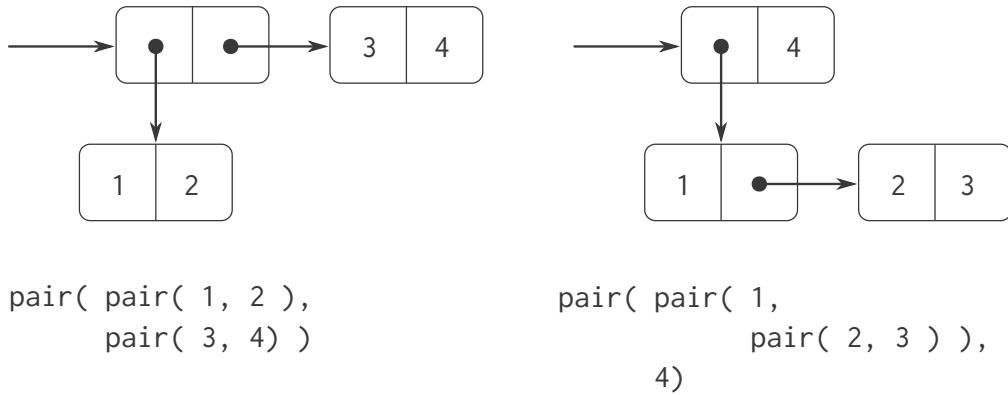


Figure 2.3: Two ways to combine 1, 2, 3, and 4 using pairs.

The ability to create pairs whose elements are pairs is the essence of list structure's importance as a representational tool. We refer to this ability as the *closure property* of pair. In general, an operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation.⁷ Closure is the key to power in any means of combination because it permits us to create *hierarchical* structures—structures made up of parts, which themselves are made up of parts, and so on.

From the outset of chapter 1, we've made essential use of closure in dealing with functions, because all but the very simplest programs rely on the fact that the elements of a combination can themselves be combinations. In this section, we take up the consequences of closure for compound data. We describe some conventional techniques for using pairs to represent sequences and trees, and we exhibit a graphics language that illustrates closure in a vivid way.⁸

⁷The use of the word ‘closure’ here comes from abstract algebra, where a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set. The Lisp community also (unfortunately) uses the word ‘closure’ to describe a totally unrelated concept: A closure is an implementation technique for representing functions with free names. We do not use the word ‘closure’ in this second sense in this book.

⁸The notion that a means of combination should satisfy closure is a straightforward idea. Unfortunately, the data combiners provided in many popular programming languages do not satisfy closure, or make closure cumbersome to exploit. In Fortran or Basic, one typically combines data elements by assembling them into arrays—but one cannot form arrays whose elements are themselves arrays. Pascal and C admit structures whose elements are structures. However, this requires that the programmer manipulate pointers explicitly, and adhere to the restriction that each field of a structure can contain only elements of a prespecified form. Unlike Lisp with its pairs, these languages have no built-in general-purpose glue that makes it easy to manipulate compound data in a uniform way. This limitation lies behind Alan Perlis's comment in his foreword to this book: ‘In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.’

2.2.1 Representing Sequences

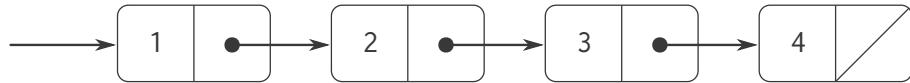


Figure 2.4: The sequence 1, 2, 3, 4 represented as a chain of pairs.

One of the useful structures we can build with pairs is a *sequence*—an ordered collection of data objects. There are, of course, many ways to represent sequences in terms of pairs. One particularly straightforward representation is illustrated in figure 2.4, where the sequence 1, 2, 3, 4 is represented as a chain of pairs. The head of each pair is the corresponding item in the chain, and the tail of the pair is the next pair in the chain. The tail of the final pair signals the end of the sequence by pointing to a distinguished value that is not a pair, represented in box-and-pointer diagrams as a diagonal line and in programs as the value of JavaScript’s value `null`. The entire sequence is constructed by nested pair operations:

```
pair(1,
    pair(2,
        pair(3,
            pair(4, null))));
```

Such a sequence of pairs is called a *list*, and our JavaScript environment provides a primitive called `list` to help in constructing lists.⁹

The above sequence could be produced by `list(1, 2, 3, 4)`. In general,

```
list(a1, a2, ..., an)
```

is equivalent to

```
pair(a1, pair(a2, pair(..., pair(an, null)...)))
```

Our interpreter prints pairs using a textual representation of box-and-pointer diagrams. The result of `pair(1, 2)` is printed as `[1, 2]`, and the data object in figure 2.4 is printed as `[1, [2, [3, [4, null]]]]`:

```
const one_through_four = list(1, 2, 3, 4);
```

We can think of `head` as selecting the first item in the list, and of `tail` as selecting the sublist consisting of all but the first item. Nested applications of `head` and `tail` can be used to extract the second, third, and subsequent items in the list. The constructor `pair` makes a list like the original one, but with an additional item at the beginning.

```
head(one_through_four);
// result: 1
```

⁹In this book, we use *list* to mean a chain of pairs terminated by the end-of-list marker. In contrast, the term *list structure* refers to any data structure made out of pairs, not just to lists.

```

tail(one_through_four);
// result: [2, [3, [4, null]]]

head(tail(one_through_four));
// result: 2

pair(10, one_through_four);
// result: [10, [1, [2, [3, [4, null]]]]]

pair(5, one_through_four);
// result: [5, [1, [2, [3, [4, null]]]]]

```

The value `null`, used to terminate the chain of pairs, can be thought of as a sequence of no elements, the *empty list*.

List operations

The use of pairs to represent sequences of elements as lists is accompanied by conventional programming techniques for manipulating lists by successively ‘tailing down’ the lists. For example, the function `list_ref` takes as arguments a list and a number n and returns the n th item of the list. It is customary to number the elements of the list beginning with 0. The method for computing `list_ref` is the following:

- For $n = 0$, `list_ref` should return the head of the list.
- Otherwise, `list_ref` should return the $(n - 1)$ st item of the tail of the list.

```

function list_ref(items, n) {
  return n === 0
    ? head(items)
    : list_ref(tail(items), n - 1);
}

const squares = list(1, 4, 9, 16, 25);
list_ref(squares, 3);
// result: 16

```

Often we tail down the whole list. To aid in this, our JavaScript environment includes a predicate `is_null`, which tests whether its argument is the empty list. The function `length`, which returns the number of items in a list, illustrates this typical pattern of use:

```

function length(items) {
  return is_null(items)
    ? 0
    : 1 + length(tail(items));
}

```

The `length` function implements a simple recursive plan. The reduction step is:

- The length of any list is 1 plus the length of the tail of the list.

This is applied successively until we reach the base case:

- The length of the empty list is 0.

We could also compute length in an iterative style:

```
function length(items) {
  function length_iter(a, count) {
    return is_null(a)
      ? count
      : length_iter(tail(a), count + 1);
  }
  return length_iter(items, 0);
}
```

Another conventional programming technique is to ‘pair up’ an answer list while tailing down a list, as in the function `append`, which takes two lists as arguments and combines their elements to make a new list:

```
append(squares, odds);
// returns: [1, [4, [9, [16, [25, [1, [3, [5, [7, null]]]]]]]]]
```

```
append(odds, squares);
```

The function `append` is also implemented using a recursive plan. To append lists `list1` and `list2`, do the following:

- If `list1` is the empty list, then the result is just `list2`.
- Otherwise, append the tail of `list1` and `list2`, and pair the head of `list1` onto the result:

```
function append(list1, list2) {
  return is_null(list1)
    ? list2
    : pair(head(list1), append(tail(list1), list2));
}
```

Exercise 2.17

Define a function `last_pair` that returns the list that contains only the last element of a given (nonempty) list:

```
last_pair(list(23, 72, 149, 34));
// result: [34, null]
```

[Solution](#)

Exercise 2.18

Define a function `reverse` that takes a list as argument and returns a list of the same elements in reverse order:

```
reverse(list(1, 4, 9, 16, 25));
// result: [25, [16, [9, [4, [1, null]]]]]
```

[Solution](#)

Exercise 2.19

Consider the change-counting program of section 1.2.2. It would be nice to be able to easily change the currency used by the program, so that we could compute the number of ways to change a British pound, for example. As the program is written, the knowledge of the currency is distributed partly into the function `first_denomination` and partly into the function `count_change` (which knows that there are five kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

We want to rewrite the function `cc` so that its second argument is a list of the values of the coins to use rather than an integer specifying which coins to use. We could then have lists that defined each kind of currency:

```
const us_coins = list(50, 25, 10, 5, 1);
const uk_coins = list(100, 50, 20, 10, 5, 2, 1, 0.5);
```

We could then call `cc` as follows:

```
cc(100, us_coins);
```

To do this will require changing the program `cc` somewhat. It will still have the same form, but it will access its second argument differently, as follows:

```
function cc(amount, coin_values) {
    return amount === 0
        ? 1
        : amount < 0 || no_more(coin_values)
        ? 0
        : cc(amount,
              except_first_denomination(coin_values))
        +
```

```

    cc(amount - first_denomination(coin_values),
       coin_values);
}

```

Define the functions `first_denomination`, `except_first_denomination`, and `no_more` in terms of primitive operations on list structures. Does the order of the list `coin_values` affect the answer produced by `cc`? Why or why not?

[Solution](#)

Exercise 2.20

In the presence of higher-order functions, it is not strictly necessary for functions to have multiple parameters; one would suffice.¹⁰ If we have a function such as `plus` that naturally requires two parameters, we could write a variant of the function to which we pass the arguments one at a time. An application of the variant to the first argument could return a function that we can then apply to the second argument, and so on. This practice—called *currying* and named after the American mathematician and logician Haskell Brooks Curry—is quite common in programming languages such as Haskell (the reader might venture a guess after whom this programming language is named) and Ocaml. In JavaScript, a curried version of `plus` looks as follows.

```

function plus_curried(x) {
  return y => x + y;
}

```

Write a function `brooks`, that takes a curried function as first argument and as second argument a list of arguments to which the curried function is then applied, one by one, in the given order. For example, the following application of `brooks` should have the same effect as the call `plus_curried(3)(4)` above.

```
brooks(plus_curried, list(3, 4));
```

While we are at it, we might as well curry the function `brooks`! Write a function `brooks_curried` that can be applied as follows, to yield the same result 7:

```
brooks_curried(list(plus_curried, 3, 4));
```

With this function `brooks_curried` what are the results of evaluating the following two statements?

```
brooks_curried(list(brooks_curried,
                     list(plus_curried, 3, 4)));
```

¹⁰Exercise 2.20 of the original textbook deals with Scheme operators that take variable numbers of arguments. This concept exists in JavaScript, but plays a less prominent role. The textbook adaptors decided to sneak in currying on this occasion.

```
brooks_curried(list(brooks_curried,
                     list(brooks_curried,
                           list(plus_curried, 3, 4))));
```

[Solution](#)

Mapping over lists

One extremely useful operation is to apply some transformation to each element in a list and generate the list of results. For instance, the following function scales each number in a list by a given factor:

```
function scale_list(items, factor) {
  return is_null(items)
    ? null
    : pair(head(items) * factor,
           scale_list(tail(items), factor));
}
```

We can abstract this general idea and capture it as a common pattern expressed as a higher-order function, just as in section 1.3. The higher-order function here is called `map`. The function `map` takes as arguments a function of one argument and a list, and returns a list of the results produced by applying the function to each element in the list:

```
function map(fun, items) {
  return is_null(items)
    ? null
    : pair(fun(head(items)),
           map(fun, tail(items)));
}

map(abs, list(-10, 2.5, -11.6, 17));

map(x => x * x, list(1, 2, 3, 4));
```

Now we can give a new definition of `scale_list` in terms of `map`:

```
function scale_list(items, factor) {
  return map(x => x * factor, items);
}
```

The function `map` is an important construct, not only because it captures a common pattern, but because it establishes a higher level of abstraction in dealing with lists. In the original definition of `scale_list`, the recursive structure of the program draws attention to the element-by-element processing of the list. Defining `scale_list` in terms of `map` suppresses that level of detail and emphasizes that scaling transforms a list of elements to a list of results. The difference between the two definitions is not that the computer is performing a different process (it

isn't) but that we think about the process differently. In effect, `map` helps establish an abstraction barrier that isolates the implementation of functions that transform lists from the details of how the elements of the list are extracted and combined. Like the barriers shown in figure 2.1, this abstraction gives us the flexibility to change the low-level details of how sequences are implemented, while preserving the conceptual framework of operations that transform sequences to sequences. section 2.2.3 expands on this use of sequences as a framework for organizing programs.

Exercise 2.21

The function `square_list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
square_list(list(1, 2, 3, 4));
// returns: [1, [4, [9, [16, null]]]]
```

Here are two different definitions of `square_list`. Complete both of them by filling in the missing expressions:

```
function square_list(items) {
    return is_null(items)
        ? null
        : pair(??, ??);
}

function square_list(items) {
    return map(??, ??);
}
```

[Solution](#)

Exercise 2.22

Louis Reasoner tries to rewrite the first `square_list` function of exercise 2.21 so that it evolves an iterative process:

```
function square_list(items) {
    function iter(things, answer) {
        return is_null(things)
            ? answer
            : iter(tail(things),
                    pair(square(head(things)),
                        answer));
    }
    return iter(items, null);
}
```

Unfortunately, defining `square_list` this way produces the answer list in the reverse order of the one desired. Why? Louis then tries to fix his bug by interchanging the arguments to `pair`:

```
function square_list(items) {
    function iter(things, answer) {
        return is_null(things)
            ? answer
            : iter(tail(things),
                  pair(answer,
                        square(head(things))));}
    return iter(items, null);
}
```

This doesn't work either. Explain.

[Solution](#)

Exercise 2.23

The function `for_each` is similar to `map`. It takes as arguments a function and a list of elements. However, rather than forming a list of the results, `for_each` just applies the function to each of the elements in turn, from left to right. The values returned by applying the function to the elements are not used at all—`for_each` is used with functions that perform an action, such as printing. For example,

```
for_each(x => display(x),
         list(57, 321, 88));
```

The value returned by the call to `for_each` (not illustrated above) can be something arbitrary, such as `true`. Give an implementation of `for_each`.

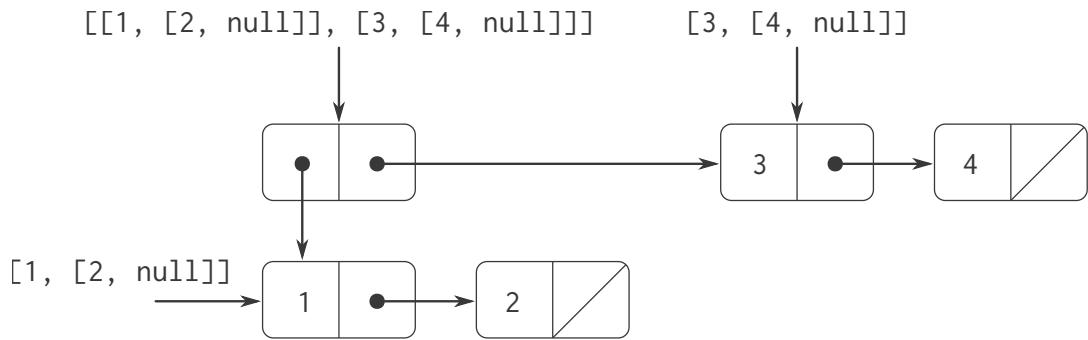
[Solution](#)

2.2.2 Hierarchical Structures

The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences. For example, we can regard the object `[[1, [2, null]], [3, [4, [5, [6, null]]]]]`, constructed by

```
pair(list(1, 2), list(3, 4));
```

as a list of three items, the first of which is itself a list, `[1, [2, null]]`. Figure 2.5 shows the representation of this structure in terms of pairs.

Figure 2.5: Structure formed by `pair(list(1, 2), list(3, 4))`.

Another way to think of sequences whose elements are sequences is as *trees*. The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees. Figure 2.6 shows the structure in Figure 2.5 viewed as a tree.

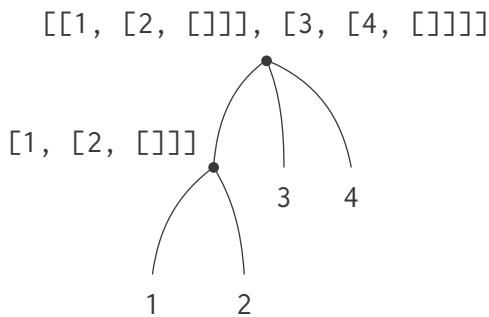


Figure 2.6: The list structure in Figure 2.5 viewed as a tree.

Recursion is a natural tool for dealing with tree structures, since we can often reduce operations on trees to operations on their branches, which reduce in turn to operations on the branches of the branches, and so on, until we reach the leaves of the tree. As an example, compare the `length` function of section 2.2.1 with the `count_leaves` function, which returns the total number of leaves of a tree:

```

const x = pair(pair(1, pair(2,null)), pair(3, pair(4,null)));
length(x);
// 3

count_leaves(x);
// 4

list(x, x);
// [[[1, [2, null]], [3, [4, null]]], 
//  [[1, [2, null]], [3, [4, null]]], null]

length(list(x, x));
// 2
    
```

```
count_leaves(list(x, x));
// 8
```

To implement `count_leaves`, recall the recursive plan for computing `length`:

- The length of a list x is 1 plus the length of the tail of x .
- The length of the empty list is 0.

The function `count_leaves` is similar. The value for the empty list is the same:

- `count_leaves` of the empty list is 0.

But in the reduction step, where we strip off the head of the list, we must take into account that the head may itself be a tree whose leaves we need to count. Thus, the appropriate reduction step is

- `count_leaves` of a tree x is `count_leaves` of the head of x plus `count_leaves` of the tail of x .

Finally, by taking heads we reach actual leaves, so we need another base case:

- `count_leaves` of a leaf is 1.

To aid in writing recursive functions on trees, our JavaScript environment provides the primitive predicate `is_pair`, which tests whether its argument is a pair. Here is the complete function:

```
function count_leaves(x) {
  return is_null(x)
    ? 0
    : ! is_pair(x)
      ? 1
      : count_leaves(head(x)) +
        count_leaves(tail(x));
}
```

Exercise 2.24

Suppose we evaluate the expression `list(1, list(2, list(3, 4)))`. Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in Figure 2.6). [Solution](#)

Exercise 2.25

Give combinations of heads and tails that will pick 7 from each of the following lists, given as printed by our interpreter:

```
[1, [3, [[5, [7, null]], [9, null]]]]
```

```
[[7, null], null]
```

```
[1,
 [
 [2,
 [
 [3,
 [
 [
 [4,
 [
 [
 [5,
 [
 [
 [6,
 [
 [
 [7,
 [
 [
 null
 ]
 ],
 null
]
]
```

[Solution](#)**Exercise 2.26**

Suppose we define x and y to be two lists:

```
const x = list(1, 2, 3);
const y = list(4, 5, 6);
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

```
append(x, y);
pair(x, y);
list(x, y);
```

[Solution](#)

Exercise 2.27

Modify your `reverse` function of exercise 2.18 to produce a `deep_reverse` function that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

```
const x = list(list(1, 2), list(3, 4));
x;
// [[1, [2, null]], [[3, [4, null]], null]]
reverse(x);
// [[3, [4, null]], [[1, [2, null]], null]]
deep_reverse(x);
// [[4, [3, null]], [[2, [1, null]], null]]
```

[Solution](#)

Exercise 2.28

Write a function `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
const x = list(list(1, 2), list(3, 4));
fringe(x);
// [1, [2, [3, [4, null]]]]
fringe(list(x, x));
// [1, [2, [3, [4, [1, [2, [3, [4, null]]]]]]]]]
```

[Solution](#)

Exercise 2.29

A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using list):

```
function make_mobile(left, right) {
    return list(left, right);
}
```

A branch is constructed from a length (which must be a number) together with a structure, which may be either a number (representing a simple weight) or another mobile:

```
function make_branch(length, structure) {
    return list(length, structure);
}
```

- a. Write the corresponding selectors `left_branch` and `right_branch`, which return the branches of a mobile, and `branch_length` and `branch_structure`, which return the components of a branch.
- b. Using your selectors, define a function `total_weight` that returns the total weight of a mobile.
- c. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.
- d. Suppose we change the representation of mobiles so that the constructors are

```
function make_mobile(left, right) {
    return pair(left, right);
}
function make_branch(length, structure) {
    return pair(length, structure);
}
```

How much do you need to change your programs to convert to the new representation?

[Solution](#)

Mapping over trees

Just as `map` is a powerful abstraction for dealing with sequences, `map` together with recursion is a powerful abstraction for dealing with trees. For instance, the `scale_tree` function, analogous to `scale_list` of section 2.2.1, takes as arguments a numeric factor and a tree whose leaves are numbers. It returns a tree of the same shape, where each number is multiplied by the factor. The recursive plan for `scale_tree` is similar to the one for `count_leaves`:

```
function scale_tree(tree, factor) {
    return is_null(tree)
        ? null
        : ! is_pair(tree)
            ? tree * factor
            : pair(scale_tree(head(tree), factor),
                  scale_tree(tail(tree), factor));
}
```

Another way to implement `scale_tree` is to regard the tree as a sequence of sub-trees and use `map`. We map over the sequence, scaling each sub-tree in turn, and return the list of results. In the base case, where the tree is a leaf, we simply multiply by the factor:

```
function scale_tree(tree, factor) {
    return map(sub_tree => is_pair(sub_tree)
                  ? scale_tree(sub_tree, factor)
                  : sub_tree * factor,
              tree);
}
```

Many tree operations can be implemented by similar combinations of sequence operations and recursion.

Exercise 2.30

Define a function `square_tree` analogous to the `square_list` function of exercise 2.21. That is, `square_tree` should behave as follows:

```
square_tree(list(1,
                 list(2, list(3, 4), 5),
                 list(6, 7)));
// result: [1, [[4, [[9, [16, null]], [25, null]]], [[36, [49, null]], null]]]
```

Define `square_tree` both directly (i.e., without using any higher-order functions) and also by using `map` and recursion.

[Solution](#)

Exercise 2.31

Abstract your answer to exercise 2.30 to produce a function `tree_map` with the property that `square_tree` could be defined as

```
function square_tree(tree) {
    return tree_map(square, tree);
}
```

[Solution](#)

Exercise 2.32

We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `[1, [2, [3, null]]]`, then the set of all subsets looks as follows:

```
[null, [[3, null], [[2, null], [[2, [3, null]], [[1, null], [[2, [3, null]], [[1, [2, null]], [[1, [2, [3, null]]], null]]]]]]]
```

Complete the following definition of a function that generates the set of subsets of a set and give a clear explanation of why it works:

```
function subsets(s) {
    if (is_null(s)) {
        return list(null);
    } else {
        const rest = subsets(tail(s));
        return append(rest, map(??, rest));
    }
}
```

[Solution](#)

2.2.3 Sequences as Conventional Interfaces

In working with compound data, we've stressed how data abstraction permits us to design programs without becoming enmeshed in the details of data representations, and how abstraction preserves for us the flexibility to experiment with alternative representations. In this section, we introduce another powerful design principle for working with data structures—the use of *conventional interfaces*.

In section 1.3 we saw how program abstractions, implemented as higher-order functions, can capture common patterns in programs that deal with numerical data. Our ability to formulate

analogous operations for working with compound data depends crucially on the style in which we manipulate our data structures. Consider, for example, the following function, analogous to the `count_leaves` function of section 2.2.2, which takes a tree as argument and computes the sum of the squares of the leaves that are odd:

```
function sum_odd_squares(tree) {
  return is_null(tree)
    ? 0
    : ! is_pair(tree)
      ? (is_odd(tree) ? square(tree) : 0)
      : sum_odd_squares(head(tree))
        +
        sum_odd_squares(tail(tree));
}
```

On the surface, this function is very different from the following one, which constructs a list of all the even Fibonacci numbers $\text{Fib}(k)$, where k is less than or equal to a given integer n :

```
function even_fibs(n) {
  function next(k) {
    if (k > n) {
      return null;
    } else {
      const f = fib(k);
      return is_even(f)
        ? pair(f, next(k + 1))
        : next(k + 1);
    }
  }
  return next(0);
}
```

Despite the fact that these two functions are structurally very different, a more abstract description of the two computations reveals a great deal of similarity. The first program

- enumerates the leaves of a tree;
- filters them, selecting the odd ones;
- squares each of the selected ones; and
- accumulates the results using `+`, starting with 0.

The second program

- enumerates the integers from 0 to n ;

- computes the Fibonacci number for each integer;
- filters them, selecting the even ones; and
- accumulates the results using `pair`, starting with the empty list.

A signal-processing engineer would find it natural to conceptualize these processes in terms of signals flowing through a cascade of stages, each of which implements part of the program plan, as shown in Figure 2.7. In `sum_odd_squares`, we begin with an *enumerator*, which generates a ‘signal’ consisting of the leaves of a given tree. This signal is passed through a *filter*, which eliminates all but the odd elements. The resulting signal is in turn passed through a *map*, which is a ‘transducer’ that applies the square function to each element. The output of the map is then fed to an *accumulator*, which combines the elements using `+`, starting from an initial 0. The plan for `even_fibs` is analogous.

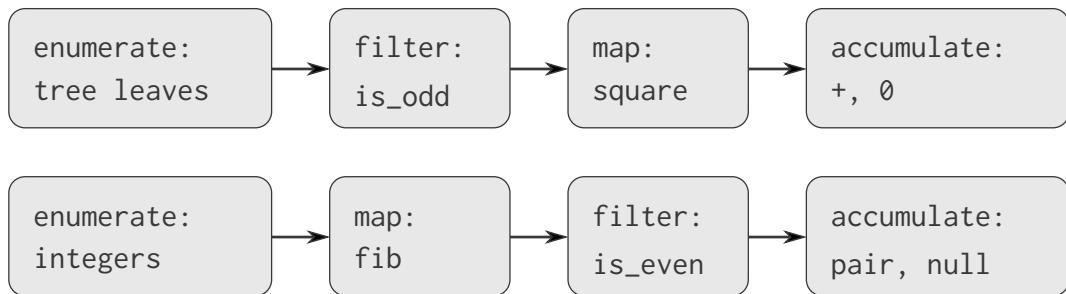


Figure 2.7: The signal-flow plans for the functions `sum_odd_squares` (top) and `even_fibs` (bottom) reveal the commonality between the two programs.

Unfortunately, the two function definitions above fail to exhibit this signal-flow structure. For instance, if we examine the `sum_odd_squares` function, we find that the enumeration is implemented partly by the `is_null` and `is_pair` tests and partly by the tree-recursive structure of the function. Similarly, the accumulation is found partly in the tests and partly in the addition used in the recursion. In general, there are no distinct parts of either function that correspond to the elements in the signal-flow description. Our two functions decompose the computations in a different way, spreading the enumeration over the program and mingling it with the map, the filter, and the accumulation. If we could organize our programs to make the signal-flow structure manifest in the functions we write, this would increase the conceptual clarity of the resulting code.

Sequence Operations

The key to organizing programs so as to more clearly reflect the signal-flow structure is to concentrate on the ‘signals’ that flow from one stage in the process to the next. If we represent these signals as lists, then we can use list operations to implement the processing at each of the stages. For instance, we can implement the mapping stages of the signal-flow diagrams using the `map` function from section 2.2.1:

```
map(square, list(1, 2, 3, 4, 5));
```

Filtering a sequence to select only those elements that satisfy a given predicate is accomplished by

```
function filter(predicate, sequence) {
    return is_null(sequence)
        ? null
        : predicate(head(sequence))
            ? pair(head(sequence),
                    filter(predicate, tail(sequence)))
            : filter(predicate, tail(sequence));
}
```

For example,

```
filter(is_odd, list(1, 2, 3, 4, 5));
```

Accumulations can be implemented by

```
function accumulate(op, initial, sequence) {
    return is_null(sequence)
        ? initial
        : op(head(sequence),
                accumulate(op, initial, tail(sequence)));
}
```

```
accumulate(plus, 0, list(1, 2, 3, 4, 5));
```

```
accumulate(times, 1, list(1, 2, 3, 4, 5));
```

```
accumulate(pair, null, list(1, 2, 3, 4, 5));
```

All that remains to implement signal-flow diagrams is to enumerate the sequence of elements to be processed. For `even_fibs`, we need to generate the sequence of integers in a given range, which we can do as follows:

```
function enumerate_interval(low, high) {
  return low > high
    ? null
    : pair(low,
              enumerate_interval(low + 1, high));
}
```

To enumerate the leaves of a tree, we can use¹¹

```
function enumerate_tree(tree) {
  return is_null(tree)
    ? null
    : ! is_pair(tree)
      ? list(tree)
      : append(enumerate_tree(head(tree)),
                enumerate_tree(tail(tree)));
}
```

Now we can reformulate `sum_odd_squares` and `even_fibs` as in the signal-flow diagrams. For `sum_odd_squares`, we enumerate the sequence of leaves of the tree, filter this to keep only the odd numbers in the sequence, square each element, and sum the results:

```
function sum_odd_squares(tree) {
  return accumulate(plus,
    0,
    map(square,
        filter(is_odd,
               enumerate_tree(tree))));
```

For `even_fibs`, we enumerate the integers from 0 to *n*, generate the Fibonacci number for each of these integers, filter the resulting sequence to keep only the even elements, and accumulate the results into a list:

```
function even_fibs(n) {
  return accumulate(pair,
    null,
    filter(is_even,
           map(fib,
               enumerate_interval(0, n))));
```

The value of expressing programs as sequence operations is that this helps us make program designs that are modular, that is, designs that are constructed by combining relatively independent pieces. We can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in flexible ways.

¹¹This is, in fact, precisely the `fringe` function from exercise 2.28. Here we've renamed it to emphasize that it is part of a family of general sequence-manipulation functions.

Modular construction is a powerful strategy for controlling complexity in engineering design. In real signal-processing applications, for example, designers regularly build systems by cascading elements selected from standardized families of filters and transducers. Similarly, sequence operations provide a library of standard program elements that we can mix and match. For instance, we can reuse pieces from the `sum_odd_squares` and `even-fibs` functions in a program that constructs a list of the squares of the first $n + 1$ Fibonacci numbers:

```
function list_fib_squares(n) {
    return accumulate(pair,
                      null,
                      map(square,
                           map(fib,
                                enumerate_interval(0, n))));
}
```

We can rearrange the pieces and use them in computing the product of the odd integers in a sequence:

```
function product_of_squares_of_odd_elements(sequence) {
    return accumulate(times,
                      1,
                      map(square,
                           filter(is_odd, sequence)));
}
```

We can also formulate conventional data-processing applications in terms of sequence operations. Suppose we have a sequence of personnel records and we want to find the salary of the highest-paid programmer. Assume that we have a selector `salary` that returns the salary of a record, and a predicate `is_programmer` that tests if a record is for a programmer. Then we can write

```
function salary_of_highest_paid_programmer(records) {
    return accumulate(math_max,
                      0,
                      map(salary,
                           filter(is_programmer, records)));
}
```

These examples give just a hint of the vast range of operations that can be expressed as sequence operations.¹²

Sequences, implemented here as lists, serve as a conventional interface that permits us to com-

¹² Richard Waters (1979) developed a program that automatically analyzes traditional Fortran programs, viewing them in terms of maps, filters, and accumulations. He found that fully 90 percent of the code in the Fortran Scientific Subroutine Package fits neatly into this paradigm. One of the reasons for the success of Lisp as a programming language is that lists provide a standard medium for expressing ordered collections so that they can be manipulated using higher-order operations. The programming language APL owes much of its power and appeal to a similar choice. In APL all data are represented as arrays, and there is a universal and convenient set of generic operators for all sorts of array operations.

bine processing modules. Additionally, when we uniformly represent structures as sequences, we have localized the data-structure dependencies in our programs to a small number of sequence operations. By changing these, we can experiment with alternative representations of sequences, while leaving the overall design of our programs intact. We will exploit this capability in section 3.5, when we generalize the sequence-processing paradigm to admit infinite sequences.

Exercise 2.33

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
function map(f, sequence) {
    return accumulate((x, y) => ?? ,
                      null, sequence);
}

function append(seq1, seq2) {
    return accumulate(pair, ??, ??);
}

function length(sequence) {
    return accumulate(??, 0, sequence);
}
```

[Solution](#)

Exercise 2.34

Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\cdots (a_nx + a_{n-1})x + \cdots + a_1)x + a_0$$

In other words, we start with a_n , multiply by x , add a_{n-1} , multiply by x , and so on, until we reach a_0 .¹³ Fill in the following template to produce a function that evaluates a polynomial

¹³According to Knuth (1981), this rule was formulated by W. G. Horner early in the nineteenth century, but the method was actually used by Newton over a hundred years earlier. Horner's rule evaluates the polynomial using fewer additions and multiplications than does the straightforward method of first computing a_nx^n , then adding $a_{n-1}x^{n-1}$, and so on. In fact, it is possible to prove that any algorithm for evaluating arbitrary polynomials

using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n .

```
function horner_eval(x, coefficient_sequence) {
    return accumulate((this_coeff, higher_terms) => ??,
                      0,
                      coefficient_sequence);
}
```

For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

```
horner_eval(2, list(1, 3, 0, 5, 0, 1));
```

[Solution](#)

Exercise 2.35

Redefine `count_leaves` from section [2.2.2](#) as an accumulation:

```
function count_leaves(t) {
    return accumulate(??, ??, map(??, ??));
}
```

[Solution](#)

Exercise 2.36

The function `accumulate_n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation function to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences

```
list(list( 1, 2, 3),
      list( 4, 5, 6),
      list( 7, 8, 9),
      list(10, 11, 12))
```

then the value of `accumulate_n(plus, 0, s)` should be the sequence `[22, [26, [30, null]]]`.

Fill in the missing expressions in the following definition of `accumulate_n`:

must use at least as many additions and multiplications as does Horner's rule, and thus Horner's rule is an optimal algorithm for polynomial evaluation. This was proved (for the number of additions) by A. M. Ostrowski in a 1954 paper that essentially founded the modern study of optimal algorithms. The analogous statement for multiplications was proved by V. Y. Pan in 1966. The book by Borodin and Munro (1975) provides an overview of these and other results about optimal algorithms.

```
function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
        ? null
        : pair(accumulate(op, init, ??),
               accumulate_n(op, init, ??));
}
```

[Solution](#)

Exercise 2.37

Suppose we represent vectors $v = (v_i)$ as sequences of numbers, and matrices $m = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

is represented as the following sequence:

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

With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

- `dot_product(v, w)` returns the sum $\sum_i v_i w_i$.
- `matrix_times_vector(m, v)` returns the vector t , where $t_i = \sum_j m_{ij} v_j$.
- `matrix_times_matrix(m, n)` returns the matrix p , where $p_{ij} = \sum_k m_{ik} n_{kj}$.
- `transpose(m)` returns the matrix n , where $n_{ij} = m_{ji}$.

We can define the dot product as¹⁴

```
function dot_product(v, w) {
    return accumulate(plus, 0,
                      accumulate_n(times, 1, list(v, w)));
}
```

Fill in the missing expressions in the following functions for computing the other matrix operations. (The function `accumulate_n` is defined in exercise 2.36.)

¹⁴This definition uses the function `accumulate_n` from exercise 2.36.

```

function matrix_times_vector(m, v) {
    return map(??, m);
}

function transpose(mat) {
    return accumulate_n(??, ??, mat);
}
function matrix_times_matrix(n, m) {
    const cols = transpose(n);
    return map(??, m);
}

```

[Solution](#)

Exercise 2.38

The accumulate function is also known as `fold_right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold_left`, which is similar to `fold_right`, except that it combines elements working in the opposite direction:

```

function fold_left(op, initial, sequence) {
    function iter(result, rest) {
        return is_null(rest)
            ? result
            : iter(op(result, head(rest)),
                  tail(rest));
    }
    return iter(initial, sequence);
}

```

What are the values of

- `fold_right(divide, 1, list(1, 2, 3));`
- `fold_left(divide, 1, list(1, 2, 3));`
- `fold_right(list, null, list(1, 2, 3));`
- `fold_left(list, null, list(1, 2, 3));`

Give a property that `op` should satisfy to guarantee that `fold_right` and `fold_left` will produce the same values for any sequence.

[Solution](#)

Exercise 2.39

Complete the following definitions of reverse (exercise 2.18) in terms of fold_right and fold_left from exercise 2.38:

```
function reverse(sequence) {
    return fold_right((x, y) => ??, null, sequence);
}

function reverse(sequence) {
    return fold_left((x, y) => ??, null, sequence);
}
```

[Solution](#)

Nested Mappings

We can extend the sequence paradigm to include many computations that are commonly expressed using nested loops.¹⁵

Consider this problem: Given a positive integer n , find all ordered pairs of distinct positive integers i and j , where $1 \leq j < i \leq n$, such that $i + j$ is prime. For example, if n is 6, then the pairs are the following:

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
<hr/>							
$i + j$	3	5	5	7	7	7	11

A natural way to organize this computation is to generate the sequence of all ordered pairs of positive integers less than or equal to n , filter to select those pairs whose sum is prime, and then, for each pair (i, j) that passes through the filter, produce the triple $(i, j, i + j)$.

Here is a way to generate the sequence of pairs: For each integer $i \leq n$, enumerate the integers $j < i$, and for each such i and j generate the pair (i, j) . In terms of sequence operations, we map along the sequence enumerate_interval(1, n). For each i in this sequence, we map along the sequence enumerate_interval(1, $i-1$). For each j in this latter sequence, we generate the pair list(i, j). This gives us a sequence of pairs for each i . Combining all the sequences for all the i (by accumulating with append) produces the required sequence of pairs:¹⁶

```
accumulate(append,
           null,
           map(i => map(j => list(i, j),
                         enumerate_interval(1, i-1)),
                enumerate_interval(1, n)));
```

¹⁵This approach to nested mappings was shown to us by David Turner, whose languages KRC and Miranda provide elegant formalisms for dealing with these constructs. The examples in this section (see also exercise 2.42) are adapted from Turner 1981. In section 3.5.3, we'll see how this approach generalizes to infinite sequences.

¹⁶We're representing a pair here as a list of two elements rather than as an ordinary pair. Thus, the 'pair' (i, j) is represented as list(i, j), not pair(i, j).

The combination of mapping and accumulating with `append` is so common in this sort of program that we will isolate it as a separate function:

```
function flatmap(f, seq) {
    return accumulate	append, null, map(f, seq));
}
```

Now filter this sequence of pairs to find those whose sum is prime. The filter predicate is called for each element of the sequence; its argument is a pair and it must extract the integers from the pair. Thus, the predicate to apply to each element in the sequence is

```
function is_prime_sum(pair) {
    return is_prime(head(pair) + head(tail(pair)));
}
```

Finally, generate the sequence of results by mapping over the filtered pairs using the following function, which constructs a triple consisting of the two elements of the pair along with their sum:

```
function make_pair_sum(pair) {
    return list(head(pair), head(tail(pair)),
               head(pair) + head(tail(pair)));
}
```

Combining all these steps yields the complete function:

```
function prime_sum_pairs(n) {
    return map(make_pair_sum,
              filter(is_prime_sum,
                     flatmap(i => map(j => list(i, j),
                                     enumerate_interval(1, i - 1)),
                             enumerate_interval(1, n))));
```

Nested mappings are also useful for sequences other than those that enumerate intervals. Suppose we wish to generate all the permutations of a set S ; that is, all the ways of ordering the items in the set. For instance, the permutations of $\{1, 2, 3\}$ are $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and $\{3, 2, 1\}$. Here is a plan for generating the permutations of S : For each item x in S , recursively generate the sequence of permutations of $S - x$,¹⁷ and adjoin x to the front of each one. This yields, for each x in S , the sequence of permutations of S that begin with x . Combining these sequences for all x gives all the permutations of S :¹⁸

¹⁷The set $S - x$ is the set of all elements of S , excluding x .

¹⁸The character sequence `//` in JavaScript code is used to introduce *comments*. Everything from `//` to the end of the line is ignored by the interpreter. In this book we don't use many comments; we try to make our programs self-documenting by using descriptive names.

```
function permutations(s) {
  return is_null(s)
    ? list(null)
    : flatmap(x => map(p => pair(x, p),
                         permutations(remove(x, s))),
              s);
}
```

Notice how this strategy reduces the problem of generating permutations of S to the problem of generating the permutations of sets with fewer elements than S . In the terminal case, we work our way down to the empty list, which represents a set of no elements. For this, we generate `list(null)`, which is a sequence with one item, namely the set with no elements. The `remove` function used in `permutations` returns all the items in a given sequence except for a given item. This can be expressed as a simple filter:

```
function remove(item, sequence) {
  return filter(x => !(x === item),
                sequence);
}
```

Exercise 2.40

Define a function `unique_pairs` that, given an integer n , generates the sequence of pairs (i, j) with $1 \leq j < i \leq n$. Use `unique_pairs` to simplify the definition of `prime_sum_pairs` given above.

[Solution](#)

Exercise 2.41

Write a function to find all ordered triples of distinct positive integers i, j , and k less than or equal to a given integer n that sum to a given integer s .

[Solution](#)

Exercise 2.42

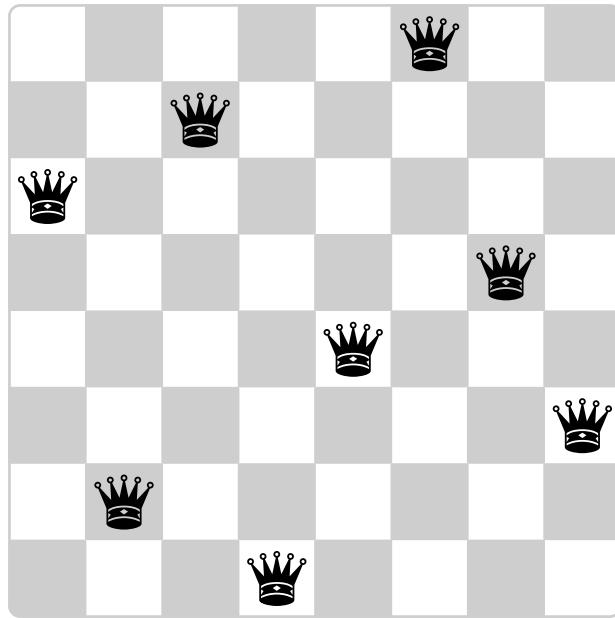


Figure 2.8: A solution to the eight-queens puzzle.

The ‘eight-queens puzzle’ asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in Figure 2.8. One way to solve the puzzle is to work across the board, placing a queen in each column. Once we have placed $k - 1$ queens, we must place the k th queen in a position where it does not check any of the queens already on the board. We can formulate this approach recursively: Assume that we have already generated the sequence of all possible ways to place $k - 1$ queens in the first $k - 1$ columns of the board. For each of these ways, generate an extended set of positions by placing a queen in each row of the k th column. Now filter these, keeping only the positions for which the queen in the k th column is safe with respect to the other queens. This produces the sequence of all ways to place k queens in the first k columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle. We implement this solution as a function `queens`, which returns a sequence of all solutions to the problem of placing n queens on an $n \times n$ chessboard. The function `queens` has an internal function `queens_cols` that returns the sequence of all ways to place queens in the first k columns of the board.

```
function queens(board_size) {
    function queen_cols(k) {
        return k === 0
            ? list(empty_board)
            : filter(
                positions => is_safe(k, positions),
                flatmap(rest_of_queens =>
                    map(new_row => adjoin_position(
                        new_row, k,
                        rest_of_queens),
```

```

        enumerate_interval(1,
                           board_size)),
        queen_cols(k - 1)));
}
return queen_cols(board_size);
}

```

In this function `rest_of_queens` is a way to place $k - 1$ queens in the first $k - 1$ columns, and `new_row` is a proposed row in which to place the queen for the k th column. Complete the program by implementing the representation for sets of board positions, including the function `adjoin_position`, which adjoins a new row-column position to a set of positions, and `empty_board`, which represents an empty set of positions. You must also write the function `is_safe`, which determines for a set of positions, whether the queen in the k th column is safe with respect to the others. (Note that we need only check whether the new queen is safe—the other queens are already guaranteed safe with respect to each other.)

[Solution](#)

Exercise 2.43

Louis Reasoner is having a terrible time doing exercise 2.42. His `queens` function seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the 6×6 case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the `flatmap`, writing it as

```

flatmap(new_row =>
    map(rest_of_queens => adjoin_position(
        new_row, k,
        rest_of_queens),
        queen_cols(k - 1)),
    enumerate_interval(1, board_size));

```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight-queens puzzle, assuming that the program in exercise 2.42 solves the puzzle in time T .

[Solution](#)

2.2.4 Example: A Picture Language

This section presents a simple language for drawing pictures that illustrates the power of data abstraction and closure, and also exploits higher-order functions in an essential way. The language is designed to make it easy to experiment with patterns such as the ones in figure 2.9, which are composed of repeated elements that are shifted and scaled.¹⁹ In this language, the data objects being combined are represented as functions rather than as list structure. Just as `pair`, which satisfies the closure property, allowed us to easily build arbitrarily complicated list structure, the operations in this language, which also satisfy the closure property, allow us to easily build arbitrarily complicated patterns.

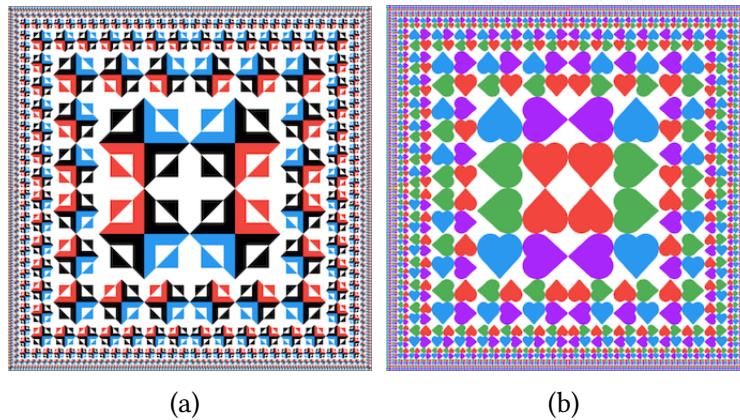


Figure 2.9: Designs generated with the picture language.

The picture language

When we began our study of programming in section 1.1, we emphasized the importance of describing a language by focusing on the language’s primitives, its means of combination, and its means of abstraction. We’ll follow that framework here.

Part of the elegance of this picture language is that there is only one kind of element, called a *painter*. A painter draws an image that is shifted and scaled to fit within a designated parallelogram-shaped frame. For example, there’s a primitive painter we’ll call `heart` that makes a heart shape, as shown in figure 2.10.

¹⁹The picture language is based on the language Peter Henderson created to construct images like M.C. Escher’s ‘Square Limit’ woodcut (see Henderson 1982). The woodcut incorporates a repeated scaled pattern, similar to the arrangements drawn using the `square_limit` function in this section.

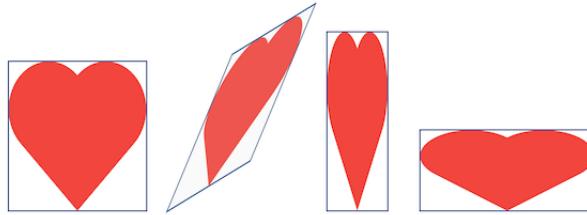


Figure 2.10: Images produced by the heart painter, with respect to four different frames. The frames, shown with thin lines, are not part of the images.

The actual shape of the drawing depends on the frame—all four images in figure 2.10 are produced by the same heart painter, but with respect to four different frames. In the following, we shall use a function `show` to display a painter in a default frame.

```
show(heart);
```

To combine images, we use various operations that construct new painters from given painters. For example, the `beside` operation takes two painters and produces a new, compound painter that draws the first painter's image in the left half of the frame and the second painter's image in the right half of the frame. Similarly, `stack` takes two painters and produces a compound painter that draws the first painter's image below the second painter's image. Some operations transform a single painter to produce a new painter. For example, `flip_vert` takes a painter and produces a painter that draws its image upside-down, and `flip_horiz` produces a painter that draws the original painter's image left-to-right reversed.

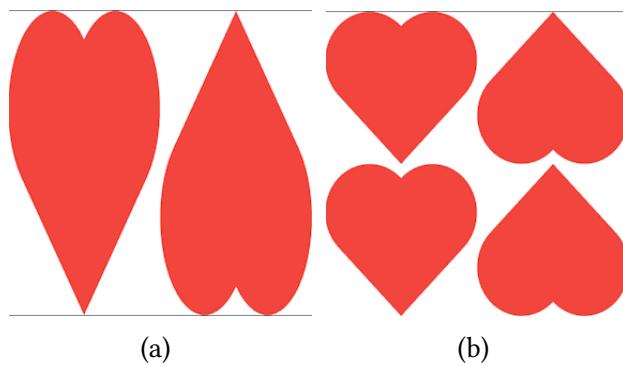


Figure 2.11: Creating a complex figure, starting from the heart painter of Figure 2.10.

Figure 2.11 shows the drawing of a painter called `heart4` that is built up in two stages starting from `heart`:

```
const heart2 = beside(heart, flip_vert(heart)); // (a)
const heart4 = stack(heart2, heart2);           // (b)
```

In building up a complex image in this manner we are exploiting the fact that painters are closed under the language's means of combination. The `beside` or `stack` of two painters is itself

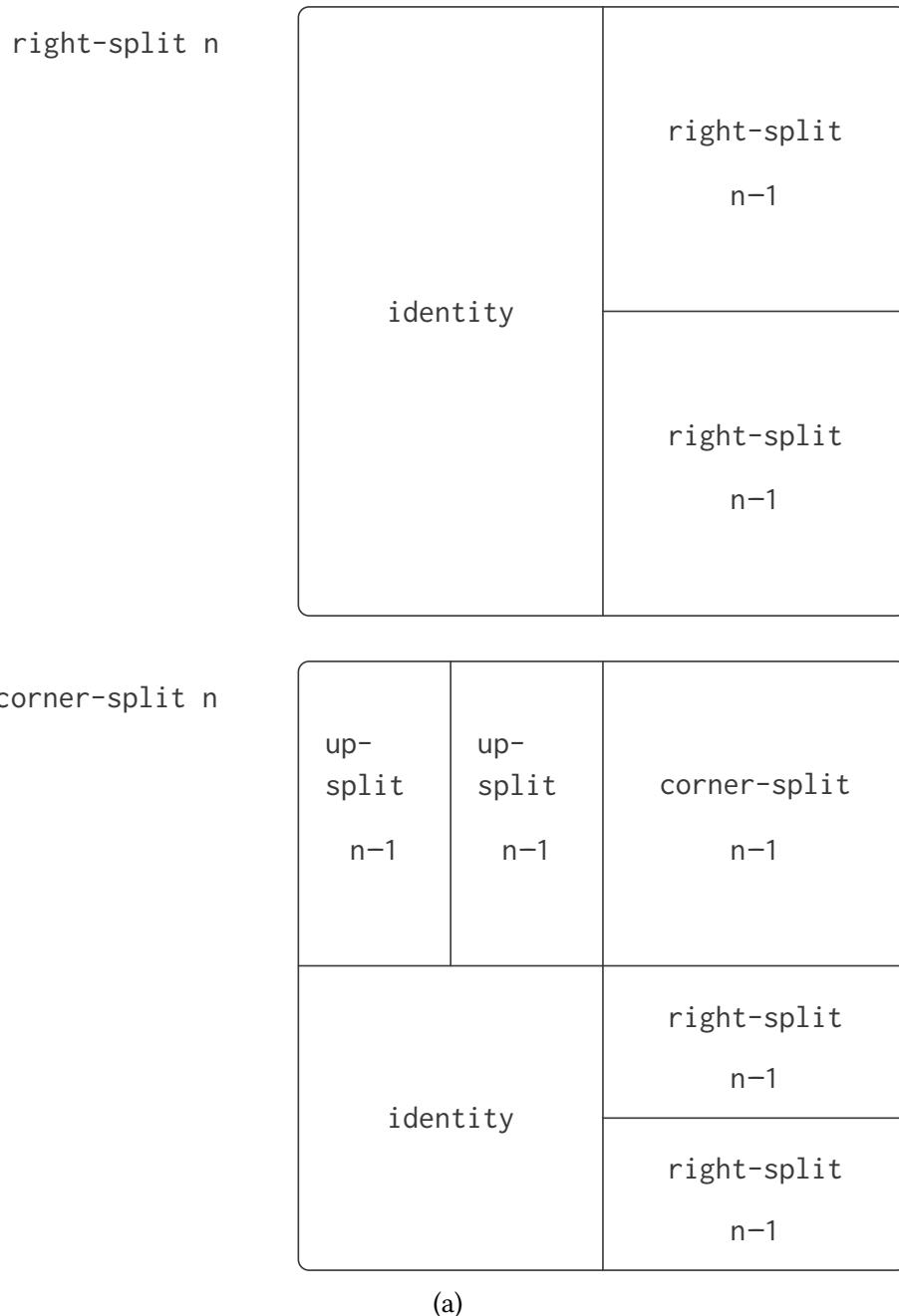
a painter; therefore, we can use it as an element in making more complex painters. As with building up list structure using `pair`, the closure of our data under the means of combination is crucial to the ability to create complex structures while using only a few operations.

Once we can combine painters, we would like to be able to abstract typical patterns of combining painters. We will implement the painter operations as JavaScript functions. This means that we don't need a special abstraction mechanism in the picture language: Since the means of combination are ordinary JavaScript functions, we automatically have the capability to do anything with painter operations that we can do with functions. For example, we can abstract the pattern in `wave4` as

```
function flipped_pairs(painter) {  
    const painter2 = beside(painter, flip_vert(painter));  
    return stack(painter2, painter2);  
}
```

and define `heart4` as an instance of this pattern:

```
const heart4 = flipped_pairs(heart);
```



(a)

Figure 2.12: Recursive plans for (a) `right_split(n)` and (b) `corner_split(n)`.

We can also define recursive operations. Here's one that makes painters split and branch towards the right as shown in figures 2.12, 2.13 and 2.14:

```
function right_split(painter, n) {
  if (n === 0) {
    return painter;
  } else {
    const smaller = right_split(painter, n - 1);
    return beside(painter, stack(smaller, smaller));
  }
}
```

We can produce balanced patterns by branching upwards as well as towards the right (see exercise 2.44 and Figure 2.12).

```
function corner_split(painter, n) {  
    if (n === 0) {  
        return painter;  
    } else {  
        const up = up_split(painter, n - 1);  
        const right = right_split(painter, n - 1);  
        const top_left = beside(up, up);  
        const bottom_right = stack(right, right);  
        const corner = corner_split(painter, n - 1);  
        return stack(beside(top_left, corner),  
                     beside(painter, bottom_right));  
    }  
}  
  
show(right_split(heart, 4));
```

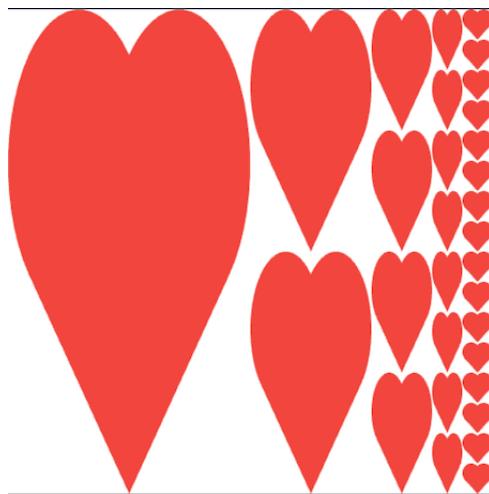


Figure 2.13: The recursive operation `right_split` applied to the painter `heart`.

```
show(corner_split(heart, 4));
```

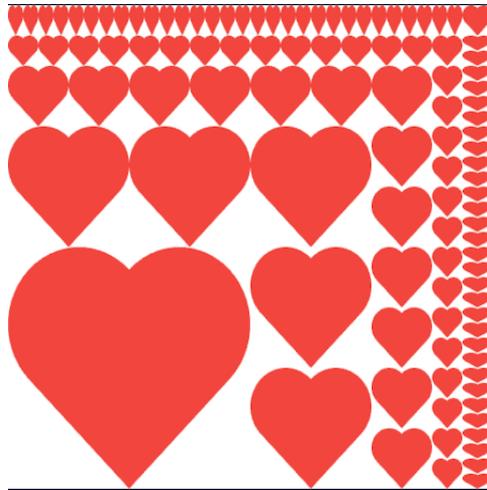


Figure 2.14: The recursive operation `corner_split` applied to the painter heart. Combining four `corner_split` figures produces symmetric `square_limit` as shown in figure 2.9.

By placing four copies of a `corner_split` appropriately, we obtain a pattern called `square_limit`, whose application to two painters is shown in figure 2.9:

```
function square_limit(painter, n) {
    const quarter = corner_split(painter, n);
    const upper_half = beside(flip_horiz(quarter), quarter);
    const lower_half = beside(turn_upside_down(quarter),
                               flip_vert(quarter));
    return stack(upper_half, lower_half);
}
```

Exercise 2.44

Define the function `up_split` used by `corner_split`. It is similar to `right_split`, except that it switches the roles of `stack` and `beside`.

[Solution](#)

Higher-order operations

In addition to abstracting patterns of combining painters, we can work at a higher level, abstracting patterns of combining painter operations. That is, we can view the painter operations as elements to manipulate and can write means of combination for these elements—functions that take painter operations as arguments and create new painter operations.

For example, `flipped_pairs` and `square_limit` each arrange four copies of a painter's image in a square pattern; they differ only in how they orient the copies. One way to abstract this pattern of painter combination is with the following function, which takes four one-argument painter operations and produces a painter operation that transforms a given painter with those

four operations and arranges the results in a square. The functions `tl`, `tr`, `bl`, and `br` are the transformations to apply to the top left copy, the top right copy, the bottom left copy, and the bottom right copy, respectively.

```
function square_of_four(tl, tr, bl, br) {
    return painter => stack(beside(tl(painter), tr(painter)),
                            beside(bl(painter), br(painter)));
}
```

Then `flipped_pairs` can be defined in terms of `square_of_four` as follows:²⁰

```
function flipped_pairs(painter) {
    const combine4 = square_of_four(turn_upside_down, flip_vert,
                                    flip_horiz, identity);
    return combine4(painter);
}
```

and `square_limit` can be expressed as²¹

```
function square_limit(painter, n) {
    const combine4 = square_of_four(flip_horiz, identity,
                                    turn_upside_down, flip_vert);
    return combine4(corner_split(painter, n));
}
```

Exercise 2.45

The functions `right_split` and `up_split` can be expressed as instances of a general splitting operation. Define a function `split` with the property that evaluating

```
const right_split = split(beside, below);
const up_split = split(below, beside);
```

produces functions `right_split` and `up_split` with the same behaviors as the ones already defined.

[Solution](#)

²⁰Equivalently, we could write

```
const flipped_pairs =
    square_of_four(turn_upside_down, flip_vert,
                  flip_horiz, identity);
```

²¹The function `turn_upside_down` rotates a painter by 180 degrees. Instead of `turn_upside_down` we could say `compose(flip_vert, flip_horiz)`, using the `compose` function from exercise 1.42.

Frames

Before we can show how to implement painters and their means of combination, we must first consider frames. A frame can be described by three vectors—an origin vector and two edge vectors. The origin vector specifies the offset of the frame's origin from some absolute origin in the plane, and the edge vectors specify the offsets of the frame's corners from its origin. If the edges are perpendicular, the frame will be rectangular. Otherwise the frame will be a more general parallelogram.

Figure 2.15 shows a frame and its associated vectors. In accordance with data abstraction, we need not be specific yet about how frames are represented, other than to say that there is a constructor `make_frame`, which takes three vectors and produces a frame, and three corresponding selectors `origin_frame`, `edge1_frame`, and `edge2_frame` (see exercise 2.47).

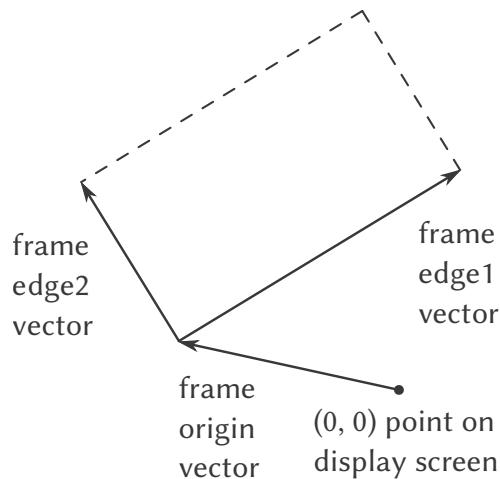


Figure 2.15: A frame is described by three vectors—an origin and two edges.

We will use coordinates in the unit square ($0 \leq x, y \leq 1$) to specify images. With each frame, we associate a *frame coordinate map*, which will be used to shift and scale images to fit the frame. The map transforms the unit square into the frame by mapping the vector $\mathbf{v} = (x, y)$ to the vector sum

$$\text{Origin(Frame)} + x \cdot \text{Edge}_1(\text{Frame}) + y \cdot \text{Edge}_2(\text{Frame})$$

For example, $(0, 0)$ is mapped to the origin of the frame, $(1, 1)$ to the vertex diagonally opposite the origin, and $(0.5, 0.5)$ to the center of the frame. We can create a frame's coordinate map with the following function:²²

²²The function `frame_coord_map` uses the vector operations described in exercise 2.46 below, which we assume have been implemented using some representation for vectors. Because of data abstraction, it doesn't matter what this vector representation is, so long as the vector operations behave correctly.

```
function frame_coord_map(frame) {
  return v => add_vect(origin_frame(frame),
                        add_vect(scale_vect(xcor_vect(v),
                                             edge1_frame(frame)),
                                 scale_vect(ycor_vect(v),
                                             edge2_frame(frame))));
}
```

Observe that applying `frame_coord_map` to a frame returns a function that, given a vector, returns a vector. If the argument vector is in the unit square, the result vector will be in the frame. For example,

```
frame_coord_map(a_frame)(make_vect(0, 0));
```

returns the same vector as

```
origin_frame(a_frame);
```

Exercise 2.46

A two-dimensional vector v running from the origin to a point can be represented as a pair consisting of an x -coordinate and a y -coordinate. Implement a data abstraction for vectors by giving a constructor `make_vect` and corresponding selectors `xcor_vect` and `ycor_vect`. In terms of your selectors and constructor, implement functions `add_vect`, `sub_vect`, and `scale_vect` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

$$\begin{aligned}(x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2) \\ (x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2) \\ s \cdot (x, y) &= (sx, sy)\end{aligned}$$

[Solution](#)

Exercise 2.47

Here are two possible constructors for frames:

```
function make_frame(origin, edge1, edge2) {
  return list(origin, edge1, edge2);
}

function make_frame(origin, edge1, edge2) {
  return pair(origin, pair(edge1, edge2));
}
```

For each constructor supply the appropriate selectors to produce an implementation for frames.

[Solution](#)

Painters

A painter is represented as a function that, given a frame as argument, draws a particular image shifted and scaled to fit the frame. That is to say, if p is a painter and f is a frame, then we produce p 's image in f by calling p with f as argument.

The details of how primitive painters are implemented depend on the particular characteristics of the graphics system and the type of image to be drawn. For instance, suppose we have a function `draw_line` that draws a line on the screen between two specified points. Then we can create painters for line drawings, such as the wave painter in figure 2.10, from lists of line segments as follows:²³

```
function segments_to_painter(segment_list) {
    return frame =>
        for_each(segment =>
            draw_line(frame_coord_map(frame)
                      (start_segment(segment)),
                      frame_coord_map(frame)
                      (end_segment(segment))),
            segment_list);
}
```

The segments are given using coordinates with respect to the unit square. For each segment in the list, the painter transforms the segment endpoints with the frame coordinate map and draws a line between the transformed points.

Representing painters as functions erects a powerful abstraction barrier in the picture language. We can create and intermix all sorts of primitive painters, based on a variety of graphics capabilities. The details of their implementation do not matter. Any function can serve as a painter, provided that it takes a frame as argument and draws something scaled to fit the frame.²⁴

²³The function `segments_to_painter` uses the representation for line segments described in exercise 2.48 below. It also uses the `for_each` function described in exercise 2.23.

²⁴For example, the heart painter of figure 2.10 was constructed from a gray-level image. For each point in a given frame, the rogers painter determines the point in the image that is mapped to it under the frame coordinate map, and shades it accordingly. By allowing different types of painters, we are capitalizing on the abstract data idea discussed in section 2.1.3, where we argued that a rational-number representation could be anything at all that satisfies an appropriate condition. Here we're using the fact that a painter can be implemented in any way at all, so long as it draws something in the designated frame. Section 2.1.3 also showed how pairs could be implemented as functions. Painters are our second example of a functional representation for data.

Exercise 2.48

A directed line segment in the plane can be represented as a pair of vectors—the vector running from the origin to the start-point of the segment, and the vector running from the origin to the end-point of the segment. Use your vector representation from exercise 2.46 to define a representation for segments with a constructor `make_segment` and selectors `start_segment` and `end_segment`.

[Solution](#)

Exercise 2.49

Use `segments_to_painter` to define the following primitive painters:

- The painter that draws the outline of the designated frame.
- The painter that draws an ‘X’ by connecting opposite corners of the frame.
- The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.

[Solution](#)

Transforming and combining painters

An operation on painters (such as `flip_vert` or `beside`) works by creating a painter that invokes the original painters with respect to frames derived from the argument frame. Thus, for example, `flip_vert` doesn’t have to know how a painter works in order to flip it—it just has to know how to turn a frame upside down: The flipped painter just uses the original painter, but in the inverted frame.

Painter operations are based on the function `transform_painter`, which takes as arguments a painter and information on how to transform a frame and produces a new painter. The transformed painter, when called on a frame, transforms the frame and calls the original painter on the transformed frame. The arguments to `transform_painter` are points (represented as vectors) that specify the corners of the new frame: When mapped into the frame,²⁵ the first

²⁵In `transform_painter`, we make use of a slight extension of the syntax of function definition expressions, compared to section 1.3.2: The body of a function definition can be a block, not just a single return expression. Such function definition expressions have the following shape:

`(parameters) => { statements }`

point specifies the new frame's origin and the other two specify the ends of its edge vectors. Thus, arguments within the unit square specify a frame contained within the original frame.

```
function transform_painter(painter, origin,
                           corner1, corner2) {
  return frame => {
    const m = frame_coord_map(frame);
    const new_origin = m(origin);
    return painter(make_frame(
      new_origin,
      sub_vect(m(corner1),
                new_origin),
      sub_vect(m(corner2),
                new_origin)));
  };
}
```

Here's how to flip painter images vertically:

```
function flip_vert(painter) {
  return transform_painter(painter,
    make_vect(0.0, 1.0), // new origin
    make_vect(1.0, 1.0), // new end of edge1
    make_vect(0.0, 0.0)); // new end of edge2
}
```

Using `transform_painter`, we can easily define new transformations. For example, we can define a painter that shrinks its image to the upper-right quarter of the frame it is given:

```
function shrink_to_upper_right(painter) {
  return transform_painter(painter,
    make_vect(0.5, 0.5),
    make_vect(1.0, 0.5),
    make_vect(0.5, 1.0));
}
```

Other transformations rotate images counterclockwise by 90 degrees²⁶

```
function rotate90(painter) {
  return transform_painter(painter,
    make_vect(1.0, 0.0),
    make_vect(1.0, 1.0),
    make_vect(0.0, 0.0));
}
```

or squash images towards the center of the frame:²⁷

²⁶The function `rotate90` is a pure rotation only for square frames, because it also stretches and shrinks the image to fit into the rotated frame.

²⁷The diamond-shaped images in figures 2.10 were created with `squash_inwards` applied to `heart`.

```
function squash_inwards(painter) {
    return transform_painter(painter,
        make_vect(0.0, 0.0),
        make_vect(0.65, 0.35),
        make_vect(0.35, 0.65));
}
```

Frame transformation is also the key to defining means of combining two or more painters. The `beside` function, for example, takes two painters, transforms them to paint in the left and right halves of an argument frame respectively, and produces a new, compound painter. When the compound painter is given a frame, it calls the first transformed painter to paint in the left half of the frame and calls the second transformed painter to paint in the right half of the frame:

```
function beside(painter1, painter2) {
    const split_point = make_vect(0.5, 0.0);
    const paint_left = transform_painter(painter1,
        make_vect(0.0, 0.0),
        split_point,
        make_vect(0.0, 1.0));
    const paint_right = transform_painter(painter2,
        split_point,
        make_vect(1.0, 0.0),
        make_vect(0.5, 1.0));
    return frame => {
        paint_left(frame);
        paint_right(frame);
    };
}
```

Observe how the painter data abstraction, and in particular the representation of painters as functions, makes `beside` easy to implement. The `beside` function need not know anything about the details of the component painters other than that each painter will draw something in its designated frame.

Exercise 2.50

Define the transformation `flip_horiz`, which flips painters horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees. [Solution](#)

Exercise 2.51

Define the `stack` operation for painters. The function `stack` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the frame

and with the second painter in the top. Define `stack` in two different ways—first by writing a function that is analogous to the `beside` function given above, and again in terms of `beside` and suitable rotation operations (from exercise 2.50). [Solution](#)

Levels of language for robust design

The picture language exercises some of the critical ideas we've introduced about abstraction with functions and data. The fundamental data abstractions, painters, are implemented using functional representations, which enables the language to handle different basic drawing capabilities in a uniform way. The means of combination satisfy the closure property, which permits us to easily build up complex designs. Finally, all the tools for abstracting functions are available to us for abstracting means of combination for painters.

We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of *stratified design*, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.

Stratified design pervades the engineering of complex systems. For example, in computer engineering, resistors and transistors are combined (and described using a language of analog circuits) to produce parts such as and-gates and or-gates, which form the primitives of a language for digital-circuit design.²⁸ These parts are combined to build processors, bus structures, and memory systems, which are in turn combined to form computers, using languages appropriate to computer architecture. Computers are combined to form distributed systems, using languages appropriate for describing network interconnections, and so on.

As a tiny example of stratification, our picture language uses primitive elements (primitive painters) that specify points and lines to provide the shapes of a painter like `heart`. The bulk of our description of the picture language focused on combining these primitives, using geometric combiners such as `beside` and `stack`. We also worked at a higher level, regarding `beside` and `stack` as primitives to be manipulated in a language whose operations, such as `square_of_four`, capture common patterns of combining geometric combiners.

Stratified design helps make programs *robust*, that is, it makes it likely that small changes in a specification will require correspondingly small changes in the program. For instance, suppose we wanted to change the image based on `heart` shown in Figure 2.9. We could work at the

²⁸Section 3.3.4 describes one such language.

lowest level to change the detailed appearance of the heart element; we could work at the middle level to change the way `corner_split` replicates the wave; we could work at the highest level to change how `square_limit` arranges the four copies of the corner. In general, each level of a stratified design provides a different vocabulary for expressing the characteristics of the system, and a different kind of ability to change it.

Exercise 2.52

Make changes to the square limit of heart shown in Figure 2.9 by working at each of the levels described above. In particular:

- a. Change the pattern constructed by `corner_split` (for example, by using only one copy of the `up_split` and `right_split` images instead of two).
- b. Modify the version of `square_limit` that uses `square_of_four` so as to assemble the corners in a different pattern.

[Solution](#)

2.3 Symbolic Data

All the compound data objects we have used so far were constructed ultimately from numbers. In this section we extend the representational capability of our language by introducing the ability to work with strings of characters as data.

2.3.1 Strings

So far, we have used strings in order to display messages, using the functions `display` and `error`, as for example in exercise 1.22). We can form compound data using strings and have lists such as

```
["a", ["b", ["c", ["d", null]]]]
list( ["Norah", 12], ["Molly", 9], ["Anna", 7],
      ["Lauren", 6], ["Charlotte", 4] )
```

Note that in order to distinguish strings from names, we surround them with double quotation marks. For example, the JavaScript expression `z` denotes the value of the name `z`, whereas the JavaScript expression `"z"` denotes a string that consists of one single character, namely the last letter in the English alphabet in lower case. JavaScript follows the common practice in natural languages, where quotation marks indicate that a word or a sentence is to be treated

literally as a string of characters. For instance, the first letter of ‘John’ is clearly ‘J.’ If we tell somebody ‘say your name aloud,’ we expect to hear that person’s name. However, if we tell somebody ‘say “your name” aloud,’ we expect to hear the words ‘your name.’ Note that we are forced to nest quotation marks to describe what somebody else might say.²⁹

Via quotation marks, we can distinguish between strings and names:

```
const a = 1;
const b = 2;

list(a, b);
// [1, [2, null]]

list("a", "b");
// ["a", ["b", null]]

list("a", b);
// ["a", [2, null]]
```

We can test whether two strings are the same, using the operator `==`.³⁰ Using `==`, we can implement a useful function called `memq`. This takes two arguments, a string and a list. If the string is not contained in the list (i.e., is not `==` to any item in the list), then `memq` returns false. Otherwise, it returns the sublist of the list beginning with the first occurrence of the string:

```
function memq(item, x) {
  return is_null(x)
    ? false
    : item === head(x)
      ? x
      : memq(item, tail(x));
}
```

For example, the value of

```
memq("apple", list("pear", "banana", "prune"));
```

is false, whereas the value of

```
memq("apple",
  list("x", list("apple", "sauce"), "y", "apple", "pear"));
```

²⁹Allowing quotation in a language wreaks havoc with the ability to reason about the language in simple terms, because it destroys the notion that equals can be substituted for equals. For example, three is one plus two, but the word ‘three’ is not the phrase ‘one plus two.’ Quotation is powerful because it gives us a way to build expressions that manipulate other expressions (as we will see when we write an interpreter in chapter 4). But allowing statements in a language that talk about other statements in that language makes it very difficult to maintain any coherent principle of what ‘equals can be substituted for equals’ should mean. For example, if we know that the evening star is the morning star, then from the statement ‘the evening star is Venus’ we can deduce ‘the morning star is Venus.’ However, given that ‘John knows that the evening star is Venus’ we cannot infer that ‘John knows that the morning star is Venus.’

³⁰We can consider two strings to be ‘the same’ if they consist of the same characters in the same order. Such a definition skirts a deep issue that we are not yet ready to address: the meaning of ‘sameness’ in a programming language. We will return to this in chapter 3 (section 3.1.3).

```
is ["apple", ["pear", null]].
```

Exercise 2.53

What would the interpreter print in response to evaluating each of the following expressions?

```
list("a", "b", "c");
list(list("george"));
tail(list(list("x1", "x2"), list("y1", "y2")));
tail(head(list(list("x1", "x2"), list("y1", "y2"))));
memq("red", list(list("red", "shoes"), list("blue", "socks")));
memq("red", list("red", "shoes", "blue", "socks"));
```

[Solution](#)

Exercise 2.54

We would like to define a function `is_equal` that checks whether two lists contain equal elements arranged in the same order. For example,

```
is_equal(list("this", "is", "a", "list"),
        list("this", "is", "a", "list"));
```

is true, but

```
is_equal(list("this", "is", "a", "list"),
        list("this", list("is", "a"), "list"));
```

is false. To be more precise, we can define `is_equal` recursively in terms of the basic `==` equality of strings by saying that `a` and `b` are equal with respect to `is_equal` if they are both strings and the strings are equal with respect to `==`, or if they are both lists such that `head(a)` is equal with respect to `is_equal` to `head(b)` and `tail(a)` is equal with respect to `is_equal` to `tail(b)`. Using this idea, implement `is_equal` as a function.³¹

[Solution](#)

Exercise 2.55

The JavaScript interpreter reads the characters after the double quotation mark `"` until it finds another double quotation mark. All characters between the two are part of the string, excluding

³¹In practice, programmers use `is_equal` to compare lists that contain numbers as well as strings. Numbers are not considered to be strings. A better definition of `is_equal` would also stipulate that if `a` and `b` are both numbers, then `a` and `b` are equal with respect to `is_equal` if they are equal with respect to `==`.

the double quotation marks, themselves. What if we want a string to contain double quotation marks? For this purpose, JavaScript also allows to use *single* quotation marks to form strings, as for example in 'say your name aloud'. Within singly-quoted strings, we can use double quotation marks, and vice versa, so 'say "your name" aloud' and "say 'your name' aloud" are valid strings that have different characters in positions 5 and 15, if we start counting at 1. Depending on the font in use, two single quotation marks might not be easily distinguishable from a double quotation mark. Can you spot which is which and work out the value of the following expression?

" ' ' == ' ' "

[Solution](#)

2.3.2 Example: Symbolic Differentiation

As an illustration of symbol manipulation and a further illustration of data abstraction, consider the design of a function that performs symbolic differentiation of algebraic expressions. We would like the function to take as arguments an algebraic expression and a variable and to return the derivative of the expression with respect to the variable. For example, if the arguments to the function are $ax^2 + bx + c$ and x , the function should return $2ax + b$. Symbolic differentiation is of special historical significance in Lisp. It was one of the motivating examples behind the development of a computer language for symbol manipulation. Furthermore, it marked the beginning of the line of research that led to the development of powerful systems for symbolic mathematical work, which are currently being used by a growing number of applied mathematicians and physicists.

In developing the symbolic-differentiation program, we will follow the same strategy of data abstraction that we followed in developing the rational-number system of section 2.1.1. That is, we will first define a differentiation algorithm that operates on abstract objects such as ‘sums,’ ‘products,’ and ‘variables’ without worrying about how these are to be represented. Only afterward will we address the representation problem.

The differentiation program with abstract data

In order to keep things simple, we will consider a very simple symbolic-differentiation program that handles expressions that are built up using only the operations of addition and multiplication with two arguments. Differentiation of any such expression can be carried out by applying the following reduction rules:

$$\frac{dc}{dx} = 0 \text{ for } c \text{ a constant or a variable different from } x$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u \left(\frac{dv}{dx} \right) + v \left(\frac{du}{dx} \right)$$

Observe that the latter two rules are recursive in nature. That is, to obtain the derivative of a sum we first find the derivatives of the terms and add them. Each of the terms may in turn be an expression that needs to be decomposed. Decomposing into smaller and smaller pieces will eventually produce pieces that are either constants or variables, whose derivatives will be either 0 or 1.

To embody these rules in a function we indulge in a little wishful thinking, as we did in designing the rational-number implementation. If we had a means for representing algebraic expressions, we should be able to tell whether an expression is a sum, a product, a constant, or a variable. We should be able to extract the parts of an expression. For a sum, for example we want to be able to extract the addend (first term) and the augend (second term). We should also be able to construct expressions from parts. Let us assume that we already have functions to implement the following selectors, constructors, and predicates:

<code>is_variable(e)</code>	Is e a variable?
<code>is_same_variable(v1, v2)</code>	Are v1 and v2 the same variable?
<code>is_sum(e)</code>	Is e a sum?
<code>addend(e)</code>	Addend of the sum e.
<code>augend(e)</code>	Augend of the sum e.
<code>make_sum(a1, a2)</code>	Construct the sum of a1 and a2.
<code>is_product(e)</code>	Is e a product?
<code>multiplier(e)</code>	Multiplier of the product e.
<code>multiplicand(e)</code>	Multiplicand of the product e.
<code>make_product(m1, m2)</code>	Construct the product of m1 and m2.

Using these, and the primitive predicate `is_number`, which identifies numbers, we can express the differentiation rules as the following function:

```
function deriv(exp, variable) {
    return is_number(exp)
        ? 0
        : is_variable(exp)
            ? (is_same_variable(exp, variable)) ? 1 : 0
            : is_sum(exp)
                ? make_sum(deriv(addend(exp), variable),
                           deriv(augend(exp), variable))
                : is_product(exp)
                    ? make_product(deriv(multiplier(exp), variable),
                                   deriv(multiplicand(exp), variable))
                    : is_number(exp)
                        ? 0
                        : error("Unknown expression type")

```

```

        deriv(augend(exp), variable))
: is_product(exp)
? make_sum(make_product(multiplier(exp),
                        deriv(multiplicand(exp),
                               variable)),
           make_product(deriv(multiplier(exp),
                               variable),
                         multiplicand(exp)))
: Error("unknown expression type in deriv",
       exp);
}

```

This `deriv` function incorporates the complete differentiation algorithm. Since it is expressed in terms of abstract data, it will work no matter how we choose to represent algebraic expressions, as long as we design a proper set of selectors and constructors. This is the issue we must address next.

Representing algebraic expressions

We can imagine many ways to use list structure to represent algebraic expressions. For example, we could use lists of symbols that mirror the usual algebraic notation, representing $ax + b$ as `list("a", "*", "x", "+", "b")`. However, it will be more convenient, if we reflect the mathematical structure of the expression in the JavaScript value representing it; that is, to represent $ax + b$ as `list("+", list("*", "a", "x"), "b")`. Then our data representation for the differentiation problem is as follows:

- The variables are strings. They are identified by the primitive predicate `is_string`:

```

function is_variable(x) {
    return is_string(x);
}

```

- Two variables are the same if the strings representing them are equal:

```

function is_same_variable(v1, v2) {
    return is_variable(v1) &&
           is_variable(v2) && v1 === v2;
}

```

- Sums and products are constructed as lists:

```

function make_sum(a1, a2) {
    return list("+", a1, a2);
}

function make_product(m1, m2) {
    return list("*", m1, m2);
}

```

- A sum is a list whose first element is the string "+":

```
function is_sum(x) {
  return is_pair(x) && head(x) === "+";
}
```

- The addend is the second item of the sum list:

```
function addend(s) {
  return head(tail(s));
}
```

- The augend is the third item of the sum list:

```
function augend(s) {
  return head(tail(tail(s)));
}
```

- A product is a list whose first element is the string "*":

```
function is_product(x) {
  return is_pair(x) && head(x) === "*";
}
```

- The multiplier is the second item of the product list:

```
function multiplier(s) {
  return head(tail(s));
}
```

- The multiplicand is the third item of the product list:

```
function multiplicand(s) {
  return head(tail(tail(s)));
}
```

Thus, we need only combine these with the algorithm as embodied by `deriv` in order to have a working symbolic-differentiation program. Let us look at some examples of its behavior:

```
deriv(list("+", "x", 3), "x");
// ["+", [1, [0, null]]]

deriv(list("*", "x", "y"), "x");
// ["*", [[ "*", ["x", [0, null]]],
//          ["*", [1, ["y", null]]], null]]

deriv(list("*", list("*", "x", "y"), list("+", "x", 3)), "x");
// [ "+",
//   [[ "*", [[ "*", ["x", ["y", null]]],
//             ["+", [1, [0, null]]], null]],
//   [[ "*", [["+",
//             [[ "*", ["x", [0, null]]], null]]], null]]]
```

```
//      [[["*", [1, ["y", null]], null]],,
//      [[["+ ", ["x", [3, null]], null] ] ],
//      null ]]]
```

The program produces answers that are correct; however, they are unsimplified. It is true that

$$\frac{d(xy)}{dx} = x \cdot 0 + 1 \cdot y$$

but we would like the program to know that $x \cdot 0 = 0$, $1 \cdot y = y$, and $0 + y = y$. The answer for the second example should have been simply y . As the third example shows, this becomes a serious issue when the expressions are complex.

Our difficulty is much like the one we encountered with the rational-number implementation: we haven't reduced answers to simplest form. To accomplish the rational-number reduction, we needed to change only the constructors and the selectors of the implementation. We can adopt a similar strategy here. We won't change `deriv` at all. Instead, we will change `make_sum` so that if both summands are numbers, `make_sum` will add them and return their sum. Also, if one of the summands is 0, then `make_sum` will return the other summand.

```
function make_sum(a1, a2) {
    return is_number_equal(a1, 0)
        ? a2
        : is_number_equal(a2, 0)
            ? a1
            : is_number(a1) && is_number(a2)
                ? a1 + a2
                : list("+", a1, a2);
}
```

This uses the function `is_number_equal`, which checks whether an expression is equal to a given number:

```
function is_number_equal(exp, num) {
    return is_number(exp) && exp === num;
}
```

Similarly, we will change `make_product` to build in the rules that 0 times anything is 0 and 1 times anything is the thing itself:

```
function make_product(m1, m2) {
    return is_number_equal(m1, 0) || is_number_equal(m2, 0)
        ? 0
        : is_number_equal(m1, 1)
            ? m2
            : is_number_equal(m2, 1)
                ? m1
                : is_number(m1) && is_number(m2)
```

```

? m1 * m2
: list("*", m1, m2);
}

```

Here is how this version works on our three examples:

```

deriv(list("+", "x", 3), "x");
// 1

deriv(list("*", "x", "y"), "x");
// "y"

deriv(list("*", list("*", "x", "y"), list("+", "x", 3)), "x");
// [ "+",
//   [[ "*", ["x", ["y", null]]],
//    [[ "*", ["y", [[ "+", ["x", [3, null]]], null]], null]] ] ]

```

Although this is quite an improvement, the third example shows that there is still a long way to go before we get a program that puts expressions into a form that we might agree is ‘simplest.’ The problem of algebraic simplification is complex because, among other reasons, a form that may be simplest for one purpose may not be for another.

Exercise 2.56

Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule for exponentiation

$$\frac{d(u^n)}{dx} = n u^{n-1} \left(\frac{du}{dx} \right)$$

by adding a new clause to the `deriv` program and defining appropriate functions `is_exp`, `base`, `exponent`, and `make_exp`. (You may use the string `"**"` to denote exponentiation.) Build in the rules that anything raised to the power 0 is 1 and anything raised to the power 1 is the thing itself.

[Solution](#)

Exercise 2.57

Extend the differentiation program to handle sums and products of arbitrary numbers of (two or more) terms. Then the last example above could be expressed as

```
deriv(list("*", "x", "y", list("+", "x", 3)), "x");
```

Try to do this by changing only the representation for sums and products, without changing the `deriv` function at all. For example, the addend of a sum would be the first term, and the augend would be the sum of the rest of the terms.

[Solution](#)

Exercise 2.58

Suppose we want to modify the differentiation program so that it works with ordinary mathematical notation, in which "+" and "*" are infix rather than prefix operators. Since the differentiation program is defined in terms of abstract data, we can modify it to work with different representations of expressions solely by changing the predicates, selectors, and constructors that define the representation of the algebraic expressions on which the differentiator is to operate.

- a. Show how to do this in order to differentiate algebraic expressions presented in infix form, as in this example:

```
list("x", "+",
     list(3, "*",
          list("x", "+",
               list("y", "+", 2))))
```

To simplify the task, assume that "+" and "*" always take two arguments and that expressions are fully parenthesized.

- b. The problem becomes substantially harder if we allow provide for avoiding unnecessary lists by assuming that multiplication is done before addition, as in this example:

```
list("x", "+", "3", "*", list("x", "+", "y", "+", 2))
```

Can you design appropriate predicates, selectors, and constructors for this notation such that our derivative program still works?

[Solution](#)

2.3.3 Example: Representing Sets

In the previous examples we built representations for two kinds of compound data objects: rational numbers and algebraic expressions. In one of these examples we had the choice of simplifying (reducing) the expressions at either construction time or selection time, but other than that the choice of a representation for these structures in terms of lists was straightforward. When we turn to the representation of sets, the choice of a representation is not so obvious. Indeed, there are a number of possible representations, and they differ significantly from one another in several ways.

Informally, a set is simply a collection of distinct objects. To give a more precise definition we can employ the method of data abstraction. That is, we define ‘set’ by specifying the operations that are to be used on sets. These are `union_set`, `intersection_set`, `is_element_of_set`, and `adjoin_set`. The function `is_element_of_set` is a predicate that determines whether a given element is a member of a set. The function `adjoin_set` takes an object and a set as arguments and returns a set that contains the elements of the original set and also the adjoined element. The function `union_set` computes the union of two sets, which is the set containing each element that appears in either argument. The function `intersection_set` computes the intersection of two sets, which is the set containing only elements that appear in both arguments. From the viewpoint of data abstraction, we are free to design any representation that implements these operations in a way consistent with the interpretations given above.³²

Sets as unordered lists

One way to represent a set is as a list of its elements in which no element appears more than once. The empty set is represented by the empty list. In this representation, `is_element_of_set` is similar to the function `memq` of section 2.3.1. It uses `is_equal` instead of `==` so that the set elements need not be primitive values:

```
function is_element_of_set(x, set) {
    return ! is_null(set) &&
           ( is_equal(x, head(set)) ||
             is_element_of_set(x, tail(set)) );
}
```

Using this, we can write `adjoin_set`. If the object to be adjoined is already in the set, we just return the set. Otherwise, we use `pair` to add the object to the list that represents the set:

```
function adjoin_set(x, set) {
    return is_element_of_set(x, set)
        ? set
        : pair(x, set);
}
```

For `intersection_set` we can use a recursive strategy. If we know how to form the intersection

³²If we want to be more formal, we can specify ‘consistent with the interpretations given above’ to mean that the operations satisfy a collection of rules such as these:

- For any set S and any object x , $\text{is_element_of_set}(x, S)$ is true (informally: ‘Adjoining an object to a set produces a set that contains the object’).
- For any sets S and T and any object x , $\text{is_element_of_set}(x, \text{union_set}(S, T))$ is equal to $\text{is_element_of_set}(x, S) \text{ || } \text{is_element_of_set}(x, T)$ (informally: ‘The elements of $\text{union}(S, T)$ are the elements that are in S or in T ’).
- For any object x , $\text{is_element_of_set}(x, \text{null})$ is false (informally: ‘No object is an element of the empty set’).

of set2 and the tail of set1, we only need to decide whether to include the head of set1 in this. But this depends on whether head(set1) is also in set2. Here is the resulting function:

```
function intersection_set(set1, set2) {
  return is_null(set1) || is_null(set2)
    ? null
    : is_element_of_set(head(set1), set2)
      ? pair(head(set1),
             intersection_set(tail(set1), set2))
      : intersection_set(tail(set1), set2);
}
```

In designing a representation, one of the issues we should be concerned with is efficiency. Consider the number of steps required by our set operations. Since they all use `is_element_of_set`, the speed of this operation has a major impact on the efficiency of the set implementation as a whole. Now, in order to check whether an object is a member of a set, `is_element_of_set` may have to scan the entire set. (In the worst case, the object turns out not to be in the set.) Hence, if the set has n elements, `is_element_of_set` might take up to n steps. Thus, the number of steps required grows as $\Theta(n)$. The number of steps required by `adjoin-set`, which uses this operation, also grows as $\Theta(n)$. For `intersection_set`, which does an `is_element_of_set` check for each element of set1, the number of steps required grows as the product of the sizes of the sets involved, or $\Theta(n^2)$ for two sets of size n . The same will be true of `union_set`.

Exercise 2.59

Implement the `union_set` operation for the unordered-list representation of sets. [Solution](#)

Exercise 2.60

We specified that a set would be represented as a list with no duplicates. Now suppose we allow duplicates. For instance, the set {1, 2, 3} could be represented as the list `list(2, 3, 2, 1, 3, 2, 2)`. Design functions `is_element_of_set`, `adjoin_set`, `union_set`, and `intersection_set` that operate on this representation. How does the efficiency of each compare with the corresponding function for the non-duplicate representation? Are there applications for which you would use this representation in preference to the non-duplicate one? [Solution](#)

Sets as ordered lists

One way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. To do this, we need some way to compare two objects so that we can say which is bigger. For example, we could compare symbols lexicographically, or we could agree on some method for assigning a unique number to an object and then compare the elements by comparing the corresponding numbers. To keep our discussion simple, we will consider only the case where the set elements are numbers, so that we can compare elements using `>` and `<`. We will represent a set of numbers by listing its elements in increasing order. Whereas our first representation above allowed us to represent the set `{1, 3, 6, 10}` by listing the elements in any order, our new representation allows only the list `list(1, 3, 6, 10)`.

One advantage of ordering shows up in `is_element_of_set`: In checking for the presence of an item, we no longer have to scan the entire set. If we reach a set element that is larger than the item we are looking for, then we know that the item is not in the set:

```
function is_element_of_set(x, set) {
    return ! is_null(set) &&
        ( x === head(set))
        ? true
        : x < head(set)
        ? false
        : is_element_of_set(x, tail(set));
}
```

How many steps does this save? In the worst case, the item we are looking for may be the largest one in the set, so the number of steps is the same as for the unordered representation. On the other hand, if we search for items of many different sizes we can expect that sometimes we will be able to stop searching at a point near the beginning of the list and that other times we will still need to examine most of the list. On the average we should expect to have to examine about half of the items in the set. Thus, the average number of steps required will be about $n/2$. This is still $\Theta(n)$ growth, but it does save us, on the average, a factor of 2 in number of steps over the previous implementation.

We obtain a more impressive speedup with `intersection_set`. In the unordered representation this operation required $\Theta(n^2)$ steps, because we performed a complete scan of `set2` for each element of `set1`. But with the ordered representation, we can use a more clever method. Begin by comparing the initial elements, `x1` and `x2`, of the two sets. If `x1` equals `x2`, then that gives an element of the intersection, and the rest of the intersection is the intersection of the tails of the two sets. Suppose, however, that `x1` is less than `x2`. Since `x2` is the smallest element in `set2`, we can immediately conclude that `x1` cannot appear anywhere in `set2` and hence is not in the intersection. Hence, the intersection is equal to the intersection of `set2` with the tail of `set1`. Similarly, if `x2` is less than `x1`, then the intersection is given by the intersection of `set1`

with the tail of set2. Here is the function:

```
function intersection_set(set1, set2) {
  if (is_null(set1) || is_null(set2)) {
    return null;
  } else {
    const x1 = head(set1);
    const x2 = head(set2);
    return x1 === x2
      ? pair(x1, intersection_set(tail(set1),
                                    tail(set2)))
      : x1 < x2
        ? intersection_set(tail(set1), set2)
        : intersection_set(set1,
                           tail(set2));
  }
}
```

To estimate the number of steps required by this process, observe that at each step we reduce the intersection problem to computing intersections of smaller sets—removing the first element from **set1** or **set2** or both. Thus, the number of steps required is at most the sum of the sizes of **set1** and **set2**, rather than the product of the sizes as with the unordered representation. This is $\Theta(n)$ growth rather than $\Theta(n^2)$ —a considerable speedup, even for sets of moderate size.

Exercise 2.61

Give an implementation of **adjoin_set** using the ordered representation. By analogy with **is_element_of_set** show how to take advantage of the ordering to produce a function that requires on the average about half as many steps as with the unordered representation. [Solution](#)

Exercise 2.62

Give a $\Theta(n)$ implementation of **union_set** for sets represented as ordered lists. [Solution](#)

Sets as binary trees

We can do better than the ordered-list representation by arranging the set elements in the form of a tree. Each node of the tree holds one element of the set, called the ‘entry’ at that node, and a link to each of two other (possibly empty) nodes. The ‘left’ link points to elements smaller than the one at the node, and the ‘right’ link to elements greater than the one at the node. Figure 2.16 shows some trees that represent the set $\{1, 3, 5, 7, 9, 11\}$. The same set may be represented by a tree in a number of different ways. The only thing we require for a valid representation is that all elements in the left subtree be smaller than the node entry and that all elements in the right subtree be larger.

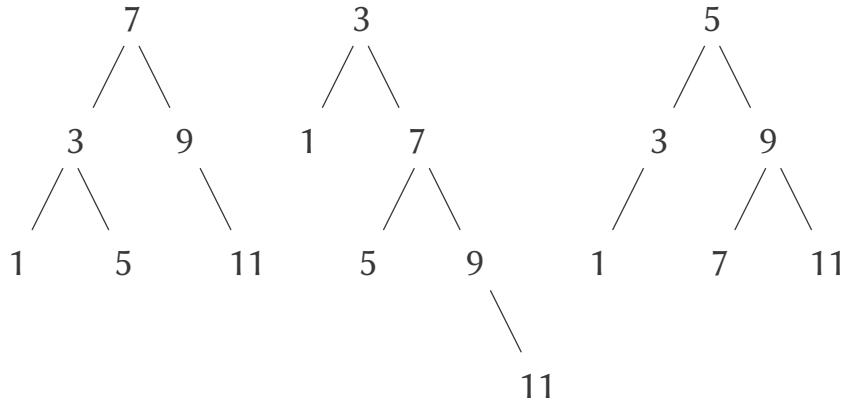


Figure 2.16: Various binary trees that represent the set $\{1, 3, 5, 7, 9, 11\}$.

The advantage of the tree representation is this: Suppose we want to check whether a number x is contained in a set. We begin by comparing x with the entry in the top node. If x is less than this, we know that we need only search the left subtree; if x is greater, we need only search the right subtree. Now, if the tree is ‘balanced,’ each of these subtrees will be about half the size of the original. Thus, in one step we have reduced the problem of searching a tree of size n to searching a tree of size $n/2$. Since the size of the tree is halved at each step, we should expect that the number of steps needed to search a tree of size n grows as $\Theta(\log n)$.³³ For large sets, this will be a significant speedup over the previous representations.

We can represent trees by using lists. Each node will be a list of three items: the entry at the node, the left subtree, and the right subtree. A left or a right subtree of the empty list will indicate that there is no subtree connected there. We can describe this representation by the following functions.³⁴

³³Halving the size of the problem at each step is the distinguishing characteristic of logarithmic growth, as we saw with the fast-exponentiation algorithm of section 1.2.4 and the half-interval search method of section 1.3.3.

³⁴We are representing sets in terms of trees, and trees in terms of lists—in effect, a data abstraction built upon a data abstraction. We can regard the functions `entry`, `left_branch`, `right_branch`, and `make_tree` as a way of isolating the abstraction of a ‘binary tree’ from the particular way we might wish to represent such a tree in terms of list structure.

```

function entry(tree) {
    return head(tree);
}
function left_branch(tree) {
    return head(tail(tree));
}
function right_branch(tree) {
    return head(tail(tail(tree)));
}
function make_tree(entry, left, right) {
    return list(entry, left, right);
}

```

Now we can write the `is_element_of_set` function using the strategy described above:

```

function is_element_of_set(x, set) {
    return ! is_null(set) &&
        ( x === entry(set) ||
        ( x < entry(set)
            ? is_element_of_set(x, left_branch(set))
            : is_element_of_set(x, right_branch(set))
        )
    );
}

```

Adjoining an item to a set is implemented similarly and also requires $\Theta(\log n)$ steps. To adjoin an item x , we compare x with the node entry to determine whether x should be added to the right or to the left branch, and having adjoined x to the appropriate branch we piece this newly constructed branch together with the original entry and the other branch. If x is equal to the entry, we just return the node. If we are asked to adjoin x to an empty tree, we generate a tree that has x as the entry and empty right and left branches. Here is the function:

```

function adjoin_set(x, set) {
    return is_null(set)
        ? make_tree(x, null, null)
        : x === entry(set)
            ? set
            : x < entry(set)
                ? make_tree(entry(set),
                    adjoin_set(x, left_branch(set)),
                    right_branch(set))
                : make_tree(entry(set),
                    left_branch(set),
                    adjoin_set(x, right_branch(set)));
}

```

The above claim that searching the tree can be performed in a logarithmic number of steps

rests on the assumption that the tree is ‘balanced,’ i.e., that the left and the right subtree of every tree have approximately the same number of elements, so that each subtree contains about half the elements of its parent. But how can we be certain that the trees we construct will be balanced? Even if we start with a balanced tree, adding elements with `adjoin_set` may produce an unbalanced result. Since the position of a newly adjoined element depends on how the element compares with the items already in the set, we can expect that if we add elements ‘randomly’ the tree will tend to be balanced on the average. But this is not a guarantee. For example, if we start with an empty set and adjoin the numbers 1 through 7 in sequence we end up with the highly unbalanced tree shown in Figure 2.17. In this tree all the left subtrees are empty, so it has no advantage over a simple ordered list. One way to solve this problem is to define an operation that transforms an arbitrary tree into a balanced tree with the same elements. Then we can perform this transformation after every few `adjoin_set` operations to keep our set in balance. There are also other ways to solve this problem, most of which involve designing new data structures for which searching and insertion both can be done in $\Theta(\log n)$ steps.³⁵

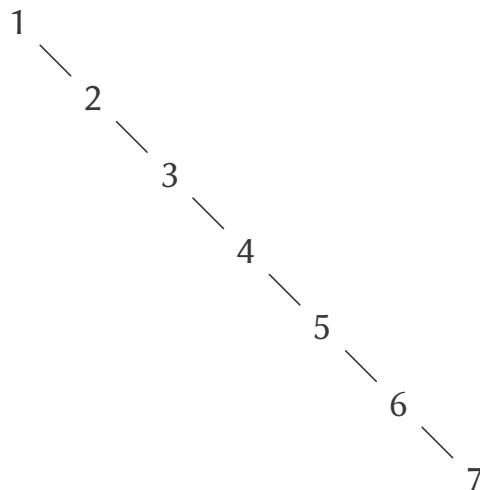


Figure 2.17: Unbalanced tree produced by adjoining 1 through 7 in sequence.

Exercise 2.63

Each of the following two functions converts a binary tree to a list.

```

function tree_to_list_1(tree) {
  return is_null(tree)
    ? null
    : append(tree_to_list_1(left_branch(tree)),
      pair(entry(tree),
        tree_to_list_1(right_branch(tree))));
  
```

³⁵Examples of such structures include *B-trees* and *red-black trees*. There is a large literature on data structures devoted to this problem. See Cormen, Leiserson, and Rivest 1990.

```

}

function tree_to_list_2(tree) {
    function copy_to_list(tree, result_list) {
        return is_null(tree)
            ? result_list
            : copy_to_list(left_branch(tree),
                pair(entry(tree),
                    copy_to_list(right_branch(tree),
                        result_list)));
    }
    return copy_to_list(tree, null);
}

```

- Do the two functions produce the same result for every tree? If not, how do the results differ? What lists do the two functions produce for the trees in Figure 2.16?
- Do the two functions have the same order of growth in the number of steps required to convert a balanced tree with n elements to a list? If not, which one grows more slowly?

[Solution](#)

Exercise 2.64

The following function `list_to_tree` converts an ordered list to a balanced binary tree. The helper function `partial_tree` takes as arguments an integer n and list of at least n elements and constructs a balanced tree containing the first n elements of the list. The result returned by `partial_tree` is a pair (formed with `pair`) whose head is the constructed tree and whose tail is the list of elements not included in the tree.

```

function list_to_tree(elements) {
    return head(partial_tree(elements, length(elements)));
}
function partial_tree(elts, n) {
    if (n === 0) {
        return pair(null, elts);
    } else {
        const left_size = math_floor((n - 1) / 2);
        const left_result = partial_tree(elts, left_size);
        const left_tree = head(left_result);
        const non_left_elts = tail(left_result);
        const right_size = n - (left_size + 1);
        const this_entry = head(non_left_elts);
        const right_result = partial_tree(tail(non_left_elts),

```

```

                    right_size);
const right_tree = head(right_result);
const remaining_elts = tail(right_result);
return pair(make_tree(this_entry,
                      left_tree,
                      right_tree),
            remaining_elts);
}
}

```

- a. Write a short paragraph explaining as clearly as you can how `partial_tree` works. Draw the tree produced by `list_to_tree` for the list `list(1, 3, 5, 7, 9, 11)`.
- b. What is the order of growth in the number of steps required by `list_to_tree` to convert a list of n elements?

[Solution](#)

Exercise 2.65

Use the results of exercises [2.63](#) and [2.64](#) to give $\Theta(n)$ implementations of `union_set` and `intersection_set` for sets implemented as (balanced) binary trees.³⁶

[Solution](#)

Sets and information retrieval

We have examined options for using lists to represent sets and have seen how the choice of representation for a data object can have a large impact on the performance of the programs that use the data. Another reason for concentrating on sets is that the techniques discussed here appear again and again in applications involving information retrieval.

Consider a data base containing a large number of individual records, such as the personnel files for a company or the transactions in an accounting system. A typical data-management system spends a large amount of time accessing or modifying the data in the records and therefore requires an efficient method for accessing records. This is done by identifying a part of each record to serve as an identifying *key*. A key can be anything that uniquely identifies the record. For a personnel file, it might be an employee's ID number. For an accounting system, it might be a transaction number. Whatever the key is, when we define the record as a data structure we should include a key selector function that retrieves the key associated with a given record.

³⁶Exercises [2.63–2.65](#) are due to Paul Hilfinger.

Now we represent the data base as a set of records. To locate the record with a given key we use a function `lookup`, which takes as arguments a key and a data base and which returns the record that has that key, or false if there is no such record. The function `lookup` is implemented in almost the same way as `is_element_of_set`. For example, if the set of records is implemented as an unordered list, we could use

```
function lookup(given_key, set_of_records) {
    return ! is_null(set_of_records) &&
        ( is_equal(given_key, key(head(set_of_records)))
        ? head(set_of_records)
        : lookup(given_key, tail(set_of_records)))
};
```

Of course, there are better ways to represent large sets than as unordered lists. Information-retrieval systems in which records have to be ‘randomly accessed’ are typically implemented by a tree-based method, such as the binary-tree representation discussed previously. In designing such a system the methodology of data abstraction can be a great help. The designer can create an initial implementation using a simple, straightforward representation such as unordered lists. This will be unsuitable for the eventual system, but it can be useful in providing a ‘quick and dirty’ data base with which to test the rest of the system. Later on, the data representation can be modified to be more sophisticated. If the data base is accessed in terms of abstract selectors and constructors, this change in representation will not require any changes to the rest of the system.

Exercise 2.66

Implement the `lookup` function for the case where the set of records is structured as a binary tree, ordered by the numerical values of the keys.

[Solution](#)

2.3.4 Example: Huffman Encoding Trees

This section provides practice in the use of list structure and data abstraction to manipulate sets and trees. The application is to methods for representing data as sequences of ones and zeros (bits). For example, the ASCII standard code used to represent text in computers encodes each character as a sequence of seven bits. Using seven bits allows us to distinguish 2^7 , or 128, possible different characters. In general, if we want to distinguish n different symbols, we will need to use $\log_2 n$ bits per symbol. If all our messages are made up of the eight symbols A, B, C, D, E, F, G, and H, we can choose a code with three bits per character, for example

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

With this code, the message

BACADAEAFABAAAGAH

is encoded as the string of 54 bits

001000010000011000100000101000001001000000000110000111

Codes such as ASCII and the A-through-H code above are known as *fixed-length* codes, because they represent each symbol in the message with the same number of bits. It is sometimes advantageous to use *variable-length* codes, in which different symbols may be represented by different numbers of bits. For example, Morse code does not use the same number of dots and dashes for each letter of the alphabet. In particular, E, the most frequent letter, is represented by a single dot. In general, if our messages are such that some symbols appear very frequently and some very rarely, we can encode data more efficiently (i.e., using fewer bits per message) if we assign shorter codes to the frequent symbols. Consider the following alternative code for the letters A through H:

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

With this code, the same message as above is encoded as the string

100010100101101100011010100100000111001111

This string contains 42 bits, so it saves more than 20% in space in comparison with the fixed-length code shown above.

One of the difficulties of using a variable-length code is knowing when you have reached the end of a symbol in reading a sequence of zeros and ones. Morse code solves this problem by using a special *separator code* (in this case, a pause) after the sequence of dots and dashes for each letter. Another solution is to design the code in such a way that no complete code for any symbol is the beginning (or *prefix*) of the code for another symbol. Such a code is called a *prefix code*. In the example above, A is encoded by 0 and B is encoded by 100, so no other symbol can have a code that begins with 0 or with 100.

In general, we can attain significant savings if we use variable-length prefix codes that take advantage of the relative frequencies of the symbols in the messages to be encoded. One particular scheme for doing this is called the Huffman encoding method, after its discoverer,

David Huffman. A Huffman code can be represented as a binary tree whose leaves are the symbols that are encoded. At each non-leaf node of the tree there is a set containing all the symbols in the leaves that lie below the node. In addition, each symbol at a leaf is assigned a weight (which is its relative frequency), and each non-leaf node contains a weight that is the sum of all the weights of the leaves lying below it. The weights are not used in the encoding or the decoding process. We will see below how they are used to help construct the tree.

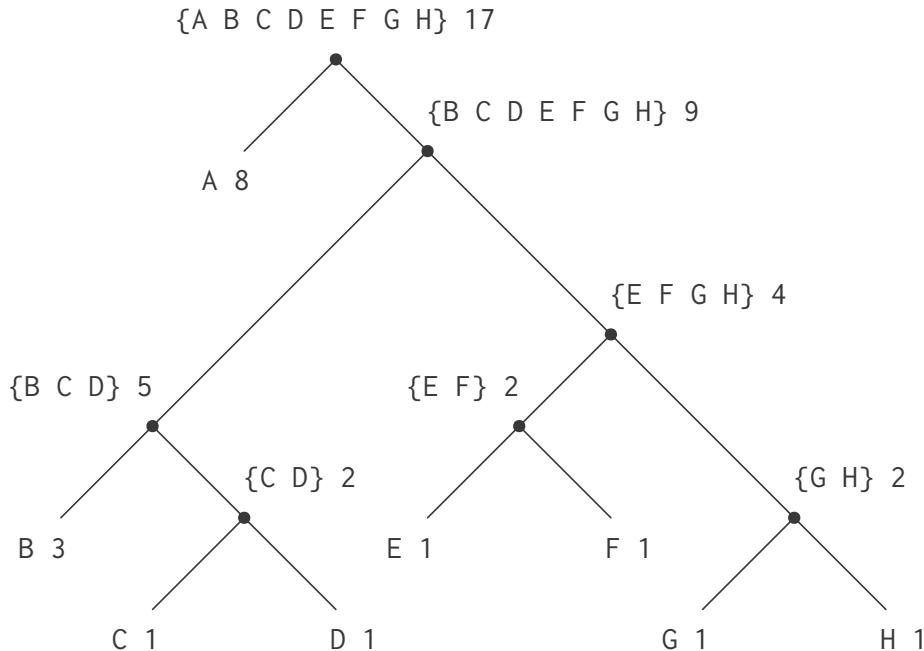


Figure 2.18: A Huffman encoding tree.

Figure 2.18 shows the Huffman tree for the A-through-H code given above. The weights at the leaves indicate that the tree was designed for messages in which A appears with relative frequency 8, B with relative frequency 3, and the other letters each with relative frequency 1.

Given a Huffman tree, we can find the encoding of any symbol by starting at the root and moving down until we reach the leaf that holds the symbol. Each time we move down a left branch we add a 0 to the code, and each time we move down a right branch we add a 1. (We decide which branch to follow by testing to see which branch either is the leaf node for the symbol or contains the symbol in its set.) For example, starting from the root of the tree in Figure 2.18, we arrive at the leaf for D by following a right branch, then a left branch, then a right branch, then a right branch; hence, the code for D is 1011.

To decode a bit sequence using a Huffman tree, we begin at the root and use the successive zeros and ones of the bit sequence to determine whether to move down the left or the right branch. Each time we come to a leaf, we have generated a new symbol in the message, at which point we start over from the root of the tree to find the next symbol. For example, suppose we are given the tree above and the sequence 10001010. Starting at the root, we move down the right branch, (since the first bit of the string is 1), then down the left branch (since the second

bit is 0), then down the left branch (since the third bit is also 0). This brings us to the leaf for B, so the first symbol of the decoded message is B. Now we start again at the root, and we make a left move because the next bit in the string is 0. This brings us to the leaf for A. Then we start again at the root with the rest of the string 1010, so we move right, left, right, left and reach C. Thus, the entire message is BAC.

Generating Huffman trees

Given an ‘alphabet’ of symbols and their relative frequencies, how do we construct the ‘best’ code? (In other words, which tree will encode messages with the fewest bits?) Huffman gave an algorithm for doing this and showed that the resulting code is indeed the best variable-length code for messages where the relative frequency of the symbols matches the frequencies with which the code was constructed. We will not prove this optimality of Huffman codes here, but we will show how Huffman trees are constructed.³⁷

The algorithm for generating a Huffman tree is very simple. The idea is to arrange the tree so that the symbols with the lowest frequency appear farthest away from the root. Begin with the set of leaf nodes, containing symbols and their frequencies, as determined by the initial data from which the code is to be constructed. Now find two leaves with the lowest weights and merge them to produce a node that has these two nodes as its left and right branches. The weight of the new node is the sum of the two weights. Remove the two leaves from the original set and replace them by this new node. Now continue this process. At each step, merge two nodes with the smallest weights, removing them from the set and replacing them with a node that has these two as its left and right branches. The process stops when there is only one node left, which is the root of the entire tree. Here is how the Huffman tree of Figure 2.18 was generated:

Initial leaves	$\{(A,8) (B,3) (C,1) (D,1) (E,1) (F,1) (G,1) (H,1)\}$
Merge	$\{(A,8) (B,3) (\{C, D\}, 2) (E, 1) (F, 1) (G,1) (H,1)\}$
Merge	$\{(A,8) (B,3) (\{C,D\},2) (\{E,F\},2) (G,1) (H,1)\}$
Merge	$\{(A,8) (B,3) (\{C,D\},2) (\{E,F\},2) (\{G,H\},2)\}$
Merge	$\{(A,8) (B,3) (\{C,D\},2) (\{E,F,G,H\},4)\}$
Merge	$\{(A,8) (\{B,C,D\},5) (\{E,F,G,H\},4)\}$
Merge	$\{(A,8) (\{B,C,D,E,F,G,H\},9)\}$
Final merge	$\{(\{A,B,C,D,E,F,G,H\},17)\}$

The algorithm does not always specify a unique tree, because there may not be unique smallest-weight nodes at each step. Also, the choice of the order in which the two nodes are merged (i.e., which will be the right branch and which will be the left branch) is arbitrary.

³⁷See Hamming 1980 for a discussion of the mathematical properties of Huffman codes.

Representing Huffman trees

In the exercises below we will work with a system that uses Huffman trees to encode and decode messages and generates Huffman trees according to the algorithm outlined above. We will begin by discussing how trees are represented.

Leaves of the tree are represented by a list consisting of the symbol leaf, the symbol at the leaf, and the weight:

```
function make_leaf(symbol, weight) {
    return list("leaf", symbol, weight);
}
function is_leaf(object) {
    return head(object) === "leaf";
}
function symbol_leaf(x) {
    return head(tail(x));
}
function weight_leaf(x) {
    return head(tail(tail(x)));
}
```

A general tree will be a list of a left branch, a right branch, a set of symbols, and a weight. The set of symbols will be simply a list of the symbols, rather than some more sophisticated set representation. When we make a tree by merging two nodes, we obtain the weight of the tree as the sum of the weights of the nodes, and the set of symbols as the union of the sets of symbols for the nodes. Since our symbol sets are represented as lists, we can form the union by using the append function we defined in section 2.2.1:

```
function make_code_tree(left,right) {
    return list(left,
               right,
               append(symbols(left), symbols(right)),
               weight(left) + weight(right));
}
```

If we make a tree in this way, we have the following selectors:

```
function left_branch(tree) {
    return head(tree);
}
function right_branch(tree) {
    return head(tail(tree));
}
function symbols(tree) {
    return is_leaf(tree)
        ? list(symbol_leaf(tree))
        : head(tail(tail(tree)));
```

```

}

function weight(tree) {
    return is_leaf(tree)
        ? weight_leaf(tree)
        : head(tail(tail(tail(tree))));
}

```

The functions `symbols` and `weight` must do something slightly different depending on whether they are called with a leaf or a general tree. These are simple examples of *generic functions* (functions that can handle more than one kind of data), which we will have much more to say about in sections 2.4 and 2.5.

The decoding function

The following function implements the decoding algorithm. It takes as arguments a list of zeros and ones, together with a Huffman tree.

```

function decode(bits, tree) {
    function decode_1(bits, current_branch) {
        if (is_null(bits)) {
            return null;
        } else {
            const next_branch = choose_branch(head(bits),
                                              current_branch);
            return is_leaf(next_branch)
                ? pair(symbol_leaf(next_branch),
                      decode_1(tail(bits), tree))
                : decode_1(tail(bits), next_branch);
        }
    }
    return decode_1(bits, tree);
}

function choose_branch(bit, branch) {
    return bit === 0
        ? left_branch(branch)
        : bit === 1
            ? right_branch(branch)
            : Error("bad bit -- choose_branch", bit);
}

```

The function `decode_1` takes two arguments: the list of remaining bits and the current position in the tree. It keeps moving ‘down’ the tree, choosing a left or a right branch according to whether the next bit in the list is a zero or a one. (This is done with the function `choose_branch`.) When it reaches a leaf, it returns the symbol at that leaf as the next symbol in the message by pairing it onto the result of decoding the rest of the message, starting at the root of the tree.

Note the error check in the final clause of `choose_branch`, which complains if the function finds something other than a zero or a one in the input data.

Sets of weighted elements

In our representation of trees, each non-leaf node contains a set of symbols, which we have represented as a simple list. However, the tree-generating algorithm discussed above requires that we also work with sets of leaves and trees, successively merging the two smallest items. Since we will be required to repeatedly find the smallest item in a set, it is convenient to use an ordered representation for this kind of set.

We will represent a set of leaves and trees as a list of elements, arranged in increasing order of weight. The following `adjoin_set` function for constructing sets is similar to the one described in exercise 2.61; however, items are compared by their weights, and the element being added to the set is never already in it.

```
function adjoin_set(x, set) {
    return is_null(set)
        ? list(x)
        : weight(x) < weight(head(set))
            ? pair(x, set)
            : pair(head(set), adjoin_set(x, tail(set)));
}
```

The following function takes a list of symbol-frequency pairs such as

```
list(list("A", 4), list("B", 2), list("C", 1), list("D", 1))
```

and constructs an initial ordered set of leaves, ready to be merged according to the Huffman algorithm:

```
function make_leaf_set(pairs) {
    if (is_null(pairs)) {
        return null;
    }
    else {
        const first_pair = head(pairs);
        return adjoin_set(
            make_leaf(head(first_pair),           // symb
                      head(tail(first_pair))), // freq
            make_leaf_set(tail(pairs)));
    }
}
```

Exercise 2.67

Define an encoding tree and a sample message:

```
const sample_tree =
  make_code_tree(make_leaf("A", 4),
    make_code_tree(make_leaf("B", 2),
      make_code_tree(make_leaf("D", 1),
        make_leaf("C", 1))));
const sample_message =
  list(0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0);
```

Use the decode function to decode the message, and give the result.

[Solution](#)

Exercise 2.68

The encode function takes as arguments a message and a tree and produces the list of bits that gives the encoded message.

```
function encode(message, tree) {
  return is_null(message)
    ? null
    : append(encode_symbol(head(message), tree),
              encode(tail(message), tree));
}
```

Write the function encode_symbol that returns the list of bits that encodes a given symbol according to a given tree. You should design encode_symbol so that it signals an error if the symbol is not in the tree at all. Test your function by encoding the result you obtained in exercise 2.67 with the sample tree and seeing whether it is the same as the original sample message.

[Solution](#)

Exercise 2.69

The following function takes as its argument a list of symbol-frequency pairs (where no symbol appears in more than one pair) and generates a Huffman encoding tree according to the Huffman algorithm.

```
function generate_huffman_tree(pairs) {
  return successive_merge(make_leaf_set(pairs));
}
```

The function make_leaf_set that transforms the list of pairs into an ordered set of leaves is given above. Write the function successive_merge using make_code_tree to successively merge the smallest-weight elements of the set until there is only one element left, which is the desired Huffman tree. (This function is slightly tricky, but not really complicated. If you find

yourself designing a complex function, then you are almost certainly doing something wrong. You can take significant advantage of the fact that we are using an ordered set representation.)

[Solution](#)

Exercise 2.70

The following eight-symbol alphabet with associated relative frequencies was designed to efficiently encode the lyrics of 1950s rock songs. (Note that the ‘symbols’ of an ‘alphabet’ need not be individual letters.)

A	2	NA	16
BOOM	1	SHA	3
GET	2	YIP	9
JOB	2	WAH	1

Use `generate_huffman_tree` (exercise 2.69) to generate a corresponding Huffman tree, and use `encode` (exercise 2.68) to encode the following message:

Get a job Sha na na na na na na na Get a job Sha na na na na na na na Wah yip yip yip yip yip yip yip yip Sha boom How many bits are required for the encoding? What is the smallest number of bits that would be needed to encode this song if we used a fixed-length code for the eight-symbol alphabet?

[Solution](#)

Exercise 2.71

Suppose we have a Huffman tree for an alphabet of n symbols, and that the relative frequencies of the symbols are $1, 2, 4, \dots, 2^{n-1}$. Sketch the tree for $n=5$; for $n=10$. In such a tree (for general n) how many bits are required to encode the most frequent symbol? the least frequent symbol?

[Solution](#)

Exercise 2.72

Consider the encoding function that you designed in exercise 2.68. What is the order of growth in the number of steps needed to encode a symbol? Be sure to include the number of steps needed to search the symbol list at each node encountered. To answer this question in general is difficult. Consider the special case where the relative frequencies of the n symbols are as described in exercise 2.71, and give the order of growth (as a function of n) of the number of

steps needed to encode the most frequent and least frequent symbols in the alphabet. [Solution](#)

2.4 Multiple Representations for Abstract Data

We have introduced data abstraction, a methodology for structuring systems in such a way that much of a program can be specified independent of the choices involved in implementing the data objects that the program manipulates. For example, we saw in section 2.1.1 how to separate the task of designing a program that uses rational numbers from the task of implementing rational numbers in terms of the computer language’s primitive mechanisms for constructing compound data. The key idea was to erect an abstraction barrier—in this case, the selectors and constructors for rational numbers (`make_rat`, `numer`, `denom`)—that isolates the way rational numbers are used from their underlying representation in terms of list structure. A similar abstraction barrier isolates the details of the functions that perform rational arithmetic (`add_rat`, `sub_rat`, `mul_rat`, and `div_rat`) from the ‘higher-level’ functions that use rational numbers. The resulting program has the structure shown in figure 2.1.

These data-abstraction barriers are powerful tools for controlling complexity. By isolating the underlying representations of data objects, we can divide the task of designing a large program into smaller tasks that can be performed separately. But this kind of data abstraction is not yet powerful enough, because it may not always make sense to speak of ‘the underlying representation’ for a data object.

For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations. To take a simple example, complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle). Sometimes rectangular form is more appropriate and sometimes polar form is more appropriate. Indeed, it is perfectly plausible to imagine a system in which complex numbers are represented in both ways, and in which the functions for manipulating complex numbers work with either representation.

More importantly, programming systems are often designed by many people working over extended periods of time, subject to requirements that change over time. In such an environment, it is simply not possible for everyone to agree in advance on choices of data representation. So in addition to the data-abstraction barriers that isolate representation from use, we need abstraction barriers that isolate different design choices from each other and permit different choices to coexist in a single program. Furthermore, since large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems *additively*, that is, without

having to redesign or reimplement these modules.

In this section, we will learn how to cope with data that may be represented in different ways by different parts of a program. This requires constructing *generic functions*—functions that can operate on data that may be represented in more than one way. Our main technique for building generic functions will be to work in terms of data objects that have *type tags*, that is, data objects that include explicit information about how they are to be processed. We will also discuss *data-directed* programming, a powerful and convenient implementation strategy for additively assembling systems with generic operations.

We begin with the simple complex-number example. We will see how type tags and data-directed style enable us to design separate rectangular and polar representations for complex numbers while maintaining the notion of an abstract ‘complex-number’ data object. We will accomplish this by defining arithmetic functions for complex numbers (`add_complex`, `sub_complex`, `mul_complex`, and `div_complex`) in terms of generic selectors that access parts of a complex number independent of how the number is represented. The resulting complex-number system, as shown in figure 2.19, contains two different kinds of abstraction barriers. The ‘horizontal’ abstraction barriers play the same role as the ones in figure 2.1. They isolate ‘higher-level’ operations from ‘lower-level’ representations. In addition, there is a ‘vertical’ barrier that gives us the ability to separately design and install alternative representations.

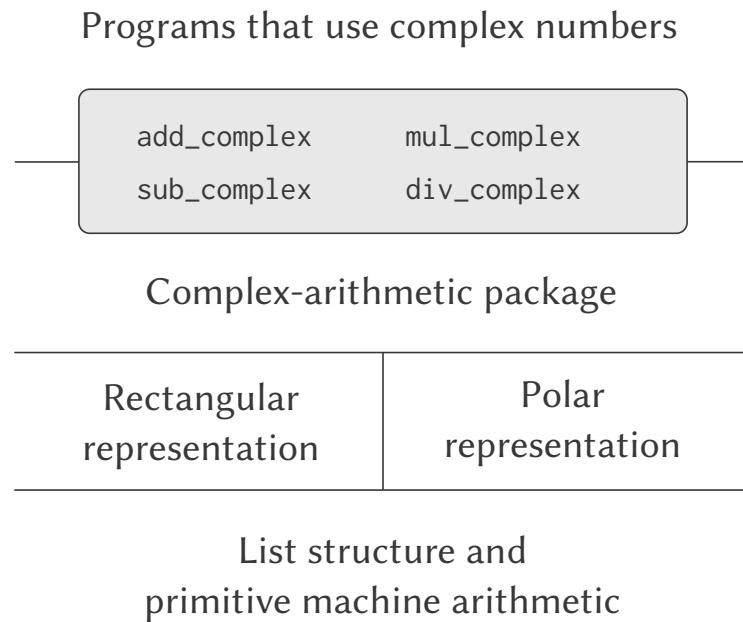


Figure 2.19: Data-abstraction barriers in the complex-number system.

In section 2.5 we will show how to use type tags and data-directed style to develop a generic arithmetic package. This provides functions (`add`, `mul`, and so on) that can be used to manipulate all sorts of ‘numbers’ and can be easily extended when a new kind of number is needed. In section 2.5.3, we’ll show how to use generic arithmetic in a system that performs symbolic

algebra.

2.4.1 Representations for Complex Numbers

We will develop a system that performs arithmetic operations on complex numbers as a simple but unrealistic example of a program that uses generic operations. We begin by discussing two plausible representations for complex numbers as ordered pairs: rectangular form (real part and imaginary part) and polar form (magnitude and angle).³⁸ Section 2.4.2 will show how both representations can be made to coexist in a single system through the use of type tags and generic operations.

Like rational numbers, complex numbers are naturally represented as ordered pairs. The set of complex numbers can be thought of as a two-dimensional space with two orthogonal axes, the ‘real’ axis and the ‘imaginary’ axis. (See figure 2.20.) From this point of view, the complex number $z = x + iy$ (where $i^2 = -1$) can be thought of as the point in the plane whose real coordinate is x and whose imaginary coordinate is y . Addition of complex numbers reduces in this representation to addition of coordinates:

$$\begin{aligned}\text{Real-part}(z_1 + z_2) &= \text{Real-part}(z_1) + \text{Real-part}(z_2) \\ \text{Imaginary-part}(z_1 + z_2) &= \text{Imaginary-part}(z_1) + \text{Imaginary-part}(z_2)\end{aligned}$$

When multiplying complex numbers, it is more natural to think in terms of representing a complex number in polar form, as a magnitude and an angle (r and A in figure 2.20). The product of two complex numbers is the vector obtained by stretching one complex number by the length of the other and then rotating it through the angle of the other:

$$\begin{aligned}\text{Magnitude}(z_1 \cdot z_2) &= \text{Magnitude}(z_1) \cdot \text{Magnitude}(z_2) \\ \text{Angle}(z_1 \cdot z_2) &= \text{Angle}(z_1) + \text{Angle}(z_2)\end{aligned}$$

Thus, there are two different representations for complex numbers, which are appropriate for different operations. Yet, from the viewpoint of someone writing a program that uses complex numbers, the principle of data abstraction suggests that all the operations for manipulating complex numbers should be available regardless of which representation is used by the computer. For example, it is often useful to be able to find the magnitude of a complex number that is specified by rectangular coordinates. Similarly, it is often useful to be able to determine

³⁸In actual computational systems, rectangular form is preferable to polar form most of the time because of roundoff errors in conversion between rectangular and polar form. This is why the complex-number example is unrealistic. Nevertheless, it provides a clear illustration of the design of a system using generic operations and a good introduction to the more substantial systems to be developed later in this chapter.

the real part of a complex number that is specified by polar coordinates.

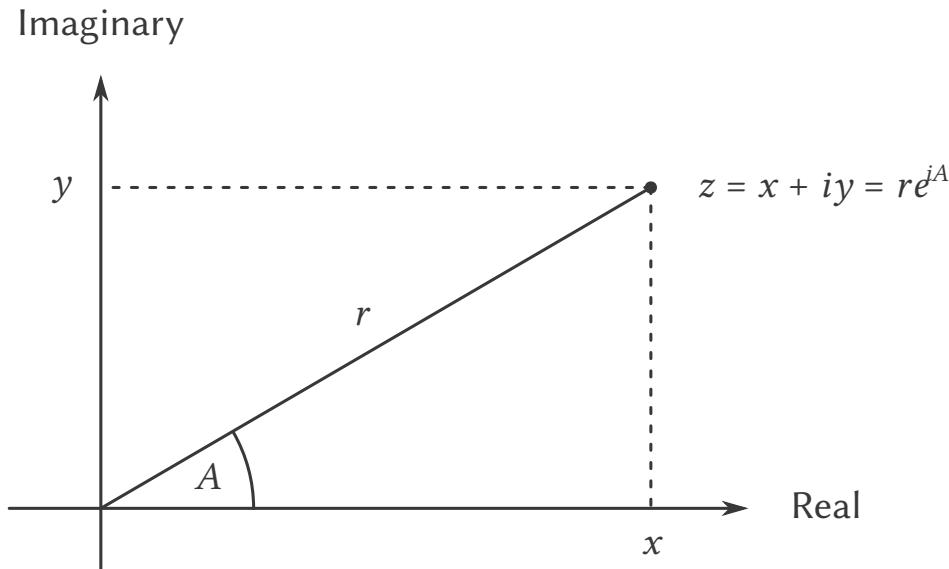


Figure 2.20: Complex numbers as points in the plane.

To design such a system, we can follow the same data-abstraction strategy we followed in designing the rational-number package in section 2.1.1. Assume that the operations on complex numbers are implemented in terms of four selectors: `real_part`, `imag_part`, `magnitude`, and `angle`. Also assume that we have two functions for constructing complex numbers: `make_from_real_imag` returns a complex number with specified real and imaginary parts, and `make_from_mag_ang` returns a complex number with specified magnitude and angle. These functions have the property that, for any complex number z , both

```
make_from_real_imag(real_part(z), imag_part(z));
```

and

```
make_from_mag_ang(magnitude(z), angle(z));
```

produce complex numbers that are equal to z .

Using these constructors and selectors, we can implement arithmetic on complex numbers using the ‘abstract data’ specified by the constructors and selectors, just as we did for rational numbers in section 2.1.1. As shown in the formulas above, we can add and subtract complex numbers in terms of real and imaginary parts while multiplying and dividing complex numbers in terms of magnitudes and angles:

```
function add_complex(z1, z2) {
    return make_from_real_imag(
        real_part(z1) + real_part(z2),
        imag_part(z1) + imag_part(z2));
}
function sub_complex(z1, z2) {
```

```

return make_from_real_imag(
    real_part(z1) - real_part(z2),
    imag_part(z1) - imag_part(z2));
}

function mul_complex(z1, z2) {
    return make_from_mag_ang(
        magnitude(z1) * magnitude(z2),
        angle(z1) + angle(z2));
}

function div_complex(z1, z2) {
    return make_from_mag_ang(
        magnitude(z1) / magnitude(z2),
        angle(z1) - angle(z2));
}

```

To complete the complex-number package, we must choose a representation and we must implement the constructors and selectors in terms of primitive numbers and primitive list structure. There are two obvious ways to do this: We can represent a complex number in ‘rectangular form’ as a pair (real part, imaginary part) or in ‘polar form’ as a pair (magnitude, angle). Which shall we choose?

In order to make the different choices concrete, imagine that there are two programmers, Ben Bitdiddle and Alyssa P. Hacker, who are independently designing representations for the complex-number system. Ben chooses to represent complex numbers in rectangular form. With this choice, selecting the real and imaginary parts of a complex number is straightforward, as is constructing a complex number with given real and imaginary parts. To find the magnitude and the angle, or to construct a complex number with a given magnitude and angle, he uses the trigonometric relations

$$\begin{aligned} x &= r \cos A & r &= \sqrt{x^2 + y^2} \\ y &= r \sin A & A &= \arctan(y, x) \end{aligned}$$

which relate the real and imaginary parts (x, y) to the magnitude and the angle (r, A).³⁹ Ben’s representation is therefore given by the following selectors and constructors:

```

function real_part(z) {
    return head(z);
}

function imag_part(z) {
    return tail(z);
}

function magnitude(z) {

```

³⁹The arctangent function referred to here, computed by JavaScript’s `math_atan2` function, is defined so as to take two arguments y and x and to return the angle whose tangent is y/x . The signs of the arguments determine the quadrant of the angle.

```

    return math_sqrt(
        square(real_part(z)) +
        square(imag_part(z)));
}
function angle(z) {
    return math_atan2(imag_part(z), real_part(z));
}
function make_from_real_imag(x, y) {
    return pair(x, y);
}
function make_from_mag_ang(r, a) {
    return pair(r * math_cos(a), r * math_sin(a));
}

```

Alyssa, in contrast, chooses to represent complex numbers in polar form. For her, selecting the magnitude and angle is straightforward, but she has to use the trigonometric relations to obtain the real and imaginary parts. Alyssa's representation is:

```

function real_part(z) {
    return magnitude(z) * math_cos(angle(z));
}
function imag_part(z) {
    return magnitude(z) * math_sin(angle(z));
}
function magnitude(z) {
    return head(z);
}
function angle(z) {
    return tail(z);
}
function make_from_real_imag(x, y) {
    return pair(math_sqrt(square(x) + square(y)),
               math_atan2(y, x));
}
function make_from_mag_ang(r, a) {
    return pair(r, a);
}

```

The discipline of data abstraction ensures that the same implementation of `add_complex`, `sub_complex`, `mul_complex`, and `div_complex` will work with either Ben's representation or Alyssa's representation.

2.4.2 Tagged data

One way to view data abstraction is as an application of the ‘principle of least commitment.’ In implementing the complex-number system in section 2.4.1, we can use either Ben’s rectangular representation or Alyssa’s polar representation. The abstraction barrier formed by the selectors and constructors permits us to defer to the last possible moment the choice of a concrete representation for our data objects and thus retain maximum flexibility in our system design.

The principle of least commitment can be carried to even further extremes. If we desire, we can maintain the ambiguity of representation even *after* we have designed the selectors and constructors, and elect to use both Ben’s representation *and* Alyssa’s representation. If both representations are included in a single system, however, we will need some way to distinguish data in polar form from data in rectangular form. Otherwise, if we were asked, for instance, to find the magnitude of the pair $(3, 4)$, we wouldn’t know whether to answer 5 (interpreting the number in rectangular form) or 3 (interpreting the number in polar form). A straightforward way to accomplish this distinction is to include a *type tag*—the symbol `rectangular` or `polar`—as part of each complex number. Then when we need to manipulate a complex number we can use the tag to decide which selector to apply.

In order to manipulate tagged data, we will assume that we have functions `type_tag` and `contents` that extract from a data object the tag and the actual contents (the polar or rectangular coordinates, in the case of a complex number). We will also postulate a function `attach_tag` that takes a tag and contents and produces a tagged data object. A straightforward way to implement this is to use ordinary list structure:

```
function attach_tag(type_tag, contents) {
    return pair(type_tag, contents);
}
function type_tag(datum) {
    return is_pair(datum)
        ? head(datum)
        : Error("bad tagged datum in type_tag", datum);
}
function contents(datum) {
    return is_pair(datum)
        ? tail(datum)
        : Error("bad tagged datum in contents", datum);
}
```

Using these functions, we can define predicates `is_rectangular` and `is_polar`, which recognize polar and rectangular numbers, respectively:

```
function is_rectangular(z) {
    return type_tag(z) === "rectangular";
}
```

```
function is_polar(z) {
    return type_tag(z) === "polar";
}
```

With type tags, Ben and Alyssa can now modify their code so that their two different representations can coexist in the same system. Whenever Ben constructs a complex number, he tags it as rectangular. Whenever Alyssa constructs a complex number, she tags it as polar. In addition, Ben and Alyssa must make sure that the names of their functions do not conflict. One way to do this is for Ben to append the suffix `rectangular` to the name of each of his representation functions and for Alyssa to append `polar` to the names of hers. Here is Ben's revised rectangular representation from section 2.4.1:

```
function real_part_rectangular(z) {
    return head(z);
}
function imag_part_rectangular(z) {
    return tail(z);
}
function magnitude_rectangular(z) {
    return math_sqrt(square(real_part_rectangular(z))
                    +
                    square(imag_part_rectangular(z)));
}
function angle_rectangular(z) {
    return math_atan(imag_part_rectangular(z),
                     real_part_rectangular(z));
}
function make_from_real_imag_rectangular(x, y) {
    return attach_tag("rectangular",
                      pair(x, y));
}
function make_from_mag_ang_rectangular(r, a) {
    return attach_tag("rectangular",
                      pair(r * math_cos(a), r * math_sin(a)));
}
```

and here is Alyssa's revised polar representation:

```
function real_part_polar(z) {
    return magnitude_polar(z) * math_cos(angle_polar(z));
}
function imag_part_polar(z) {
    return magnitude_polar(z) * math_sin(angle_polar(z));
}
function magnitude_polar(z) {
    return head(z);
}
```

```

function angle_polar(z) {
  return tail(z);
}
function make_from_real_imag_polar(x, y) {
  return attach_tag("polar",
                    pair(math_sqrt(square(x) + square(y)),
                         math_atan(y, z)));
}
function make_from_mag_ang_polar(r, a) {
  return attach_tag("polar",
                    pair(r, a));
}

```

Each generic selector is implemented as a function that checks the tag of its argument and calls the appropriate function for handling data of that type. For example, to obtain the real part of a complex number, `real_part` examines the tag to determine whether to use Ben's `real_part_rectangular` or Alyssa's `real_part_polar`. In either case, we use `contents` to extract the bare, untagged datum and send this to the rectangular or polar function as required:

```

function real_part(z) {
  return is_rectangular(z)
    ? real_part_rectangular(contents(z))
    : is_polar(z)
      ? real_part_polar(contents(z))
      : Error("Unknown type in real_part", z);
}
function imag_part(z) {
  return is_rectangular(z)
    ? imag_part_rectangular(contents(z))
    : is_polar(z)
      ? imag_part_polar(contents(z))
      : Error("Unknown type in imag_part", z);
}
function magnitude(z) {
  return is_rectangular(z)
    ? magnitude_rectangular(contents(z))
    : is_polar(z)
      ? magnitude_polar(contents(z))
      : Error("Unknown type in magnitude", z);
}
function angle(z) {
  return is_rectangular(z)
    ? angle_rectangular(contents(z))
    : is_polar(z)
      ? angle_polar(contents(z))
      : Error("Unknown type in angle", z);
}

```

To implement the complex-number arithmetic operations, we can use the same functions `add_complex`, `sub_complex`, `mul_complex`, and `div_complex` from section 2.4.1, because the selectors they call are generic, and so will work with either representation. For example, the function `add_complex` is still

```
function add_complex(z1, z2) {
    return make_from_real_imag(
        real_part(z1) + real_part(z2),
        imag_part(z1) + imag_part(z2));
}
```

Finally, we must choose whether to construct complex numbers using Ben’s representation or Alyssa’s representation. One reasonable choice is to construct rectangular numbers whenever we have real and imaginary parts and to construct polar numbers whenever we have magnitudes and angles:

```
function make_from_real_imag(x, y) {
    return make_from_real_imag_rectangular(x, y);
}
function make_from_mag_ang(r, a) {
    return make_from_mag_ang_polar(r, a);
}
```

Programs that use complex numbers

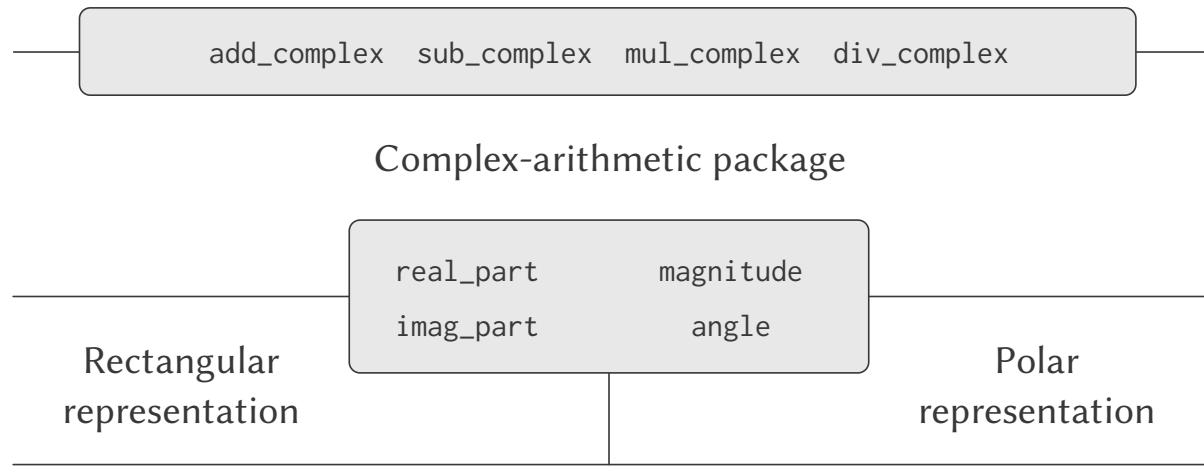


Figure 2.21: Structure of the generic complex-arithmetic system.

The resulting complex-number system has the structure shown in figure 2.21. The system has been decomposed into three relatively independent parts: the complex-number-arithmetic operations, Alyssa’s polar implementation, and Ben’s rectangular implementation. The polar and rectangular implementations could have been written by Ben and Alyssa working separately, and both of these can be used as underlying representations by a third programmer

implementing the complex-arithmetic functions in terms of the abstract constructor/selector interface.

Since each data object is tagged with its type, the selectors operate on the data in a generic manner. That is, each selector is defined to have a behavior that depends upon the particular type of data it is applied to. Notice the general mechanism for interfacing the separate representations: Within a given representation implementation (say, Alyssa's polar package) a complex number is an untyped pair (magnitude, angle). When a generic selector operates on a number of `polar` type, it strips off the tag and passes the contents on to Alyssa's code. Conversely, when Alyssa constructs a number for general use, she tags it with a type so that it can be appropriately recognized by the higher-level functions. This discipline of stripping off and attaching tags as data objects are passed from level to level can be an important organizational strategy, as we shall see in section [2.5](#).

2.4.3 Data-Directed Programming and Additivity

The general strategy of checking the type of a datum and calling an appropriate function is called *dispatching on type*. This is a powerful strategy for obtaining modularity in system design. On the other hand, implementing the dispatch as in section [2.4.2](#) has two significant weaknesses. One weakness is that the generic interface functions (`real_part`, `imag_part`, `magnitude`, and `angle`) must know about all the different representations. For instance, suppose we wanted to incorporate a new representation for complex numbers into our complex-number system. We would need to identify this new representation with a type, and then add a clause to each of the generic interface functions to check for the new type and apply the appropriate selector for that representation.

Another weakness of the technique is that even though the individual representations can be designed separately, we must guarantee that no two functions in the entire system have the same name. This is why Ben and Alyssa had to change the names of their original functions from section [2.4.1](#).

The issue underlying both of these weaknesses is that the technique for implementing generic interfaces is not *additive*. The person implementing the generic selector functions must modify those functions each time a new representation is installed, and the people interfacing the individual representations must modify their code to avoid name conflicts. In each of these cases, the changes that must be made to the code are straightforward, but they must be made nonetheless, and this is a source of inconvenience and error. This is not much of a problem for the complex-number system as it stands, but suppose there were not two but hundreds of different representations for complex numbers. And suppose that there were many generic selectors to be maintained in the abstract-data interface. Suppose, in fact, that no one programmer knew all the interface functions or all the representations. The problem is real and must

be addressed in such programs as large-scale data-base-management systems.

What we need is a means for modularizing the system design even further. This is provided by the programming technique known as *data-directed programming*. To understand how data-directed programming works, begin with the observation that whenever we deal with a set of generic operations that are common to a set of different types we are, in effect, dealing with a two-dimensional table that contains the possible operations on one axis and the possible types on the other axis. The entries in the table are the functions that implement each operation for each type of argument presented. In the complex-number system developed in the previous section, the correspondence between operation name, data type, and actual function was spread out among the various conditional clauses in the generic interface functions. But the same information could have been organized in a table, as shown in figure 2.22.

Data-directed programming is the technique of designing programs to work with such a table directly. Previously, we implemented the mechanism that interfaces the complex-arithmetic code with the two representation packages as a set of functions that each perform an explicit dispatch on type. Here we will implement the interface as a single function that looks up the combination of the operation name and argument type in the table to find the correct function to apply, and then applies it to the contents of the argument. If we do this, then to add a new representation package to the system we need not change any existing functions; we need only add new entries to the table.

Operations	Types	
	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

Figure 2.22: Table of operations for the complex-number system.

To implement this plan, assume that we have two functions, put and get, for manipulating the operation-and-type table:

- `put(<op>, <type>, <item>)` installs the `<item>` in the table, indexed by the `<op>` and the `<type>`.
- `get(<op>, <type>)` looks up the `<op>, <type>` entry in the table and returns the item found there. If no item is found, `get` returns `undefined`.

For now, we can assume that put and get are included in our language. In chapter 3 (section 3.3.3) we will see how to implement these and other operations for manipulating tables.

Here is how data-directed programming can be used in the complex-number system. Ben, who developed the rectangular representation, implements his code just as he did originally. He defines a collection of functions, or a *package*, and interfaces these to the rest of the system by adding entries to the table that tell the system how to operate on rectangular numbers. This is accomplished by calling the following function:

```

function install_rectangular_package() {
    function real_part(z) { return head(z); }
    function imag_part(z) { return tail(z); }
    function make_from_real_imag(x, y) { return pair(x, y); }
    function magnitude(z) {
        return math_sqrt(square(real_part(z)) +
                        square(imag_part(z)));
    }
    function angle(z) {
        return math_atan(imag_part(z), real_part(z));
    }
    function make_from_mag_ang(r, a) {
        return pair(r * math_cos(a), r * math_sin(a));
    }
    // interface to the rest of the system
    function tag(x) {
        return attach_tag("rectangular", x);
    }
    put("real_part", list("rectangular"), real_part);
    put("imag_part", list("rectangular"), imag_part);
    put("magnitude", list("rectangular"), magnitude);
    put("angle", list("rectangular"), angle);
    put("make_from_real_imag", "rectangular",
        (x, y) => tag(make_from_real_imag(x, y)));
    put("make_from_mag_ang", "rectangular",
        (r, a) => tag(make_from_mag_ang(r, a)));
    return "done";
}

install_rectangular_package();

```

Notice that the internal functions here are the same functions from section 2.4.1 that Ben wrote when he was working in isolation. No changes are necessary in order to interface them to the rest of the system. Moreover, since these function definitions are internal to the installation function, Ben needn't worry about name conflicts with other functions outside the rectangular package. To interface these to the rest of the system, Ben installs his `real_part` function under the operation name `real_part` and the type `list("rectangular")`, and similarly for the other

selectors.⁴⁰ The interface also defines the constructors to be used by the external system.⁴¹ These are identical to Ben's internally defined constructors, except that they attach the tag. Alyssa's polar package is analogous:

```

function install_polar_package() {
    // internal functions
    function magnitude(z) { return head(z); }
    function angle(z) { return tail(z); }
    function make_from_mag_ang(r, a) { return pair(r, a); }
    function real_part(z) {
        return magnitude(z) * math_cos(angle(z));
    }
    function imag_part(z) {
        return magnitude(z) * math_sin(angle(z));
    }
    function make_from_real_imag(x, y) {
        return pair(math_sqrt(square(x) + square(y)),
                    math_atan(y, x));
    }

    // interface to the rest of the system
    function tag(x) { return attach_tag("polar", x); }
    put("real_part", list("polar"), real_part);
    put("imag_part", list("polar"), imag_part);
    put("magnitude", list("polar"), magnitude);
    put("angle", list("polar"), angle);
    put("make_from_real_imag", "polar",
        (x, y) => tag(make_from_real_imag(x, y)));
    put("make_from_mag_ang", "polar",
        (r, a) => tag(make_from_mag_ang(r, a)));
    return "done";
}

install_polar_package();

```

Even though Ben and Alyssa both still use their original functions defined with the same names as each other's (e.g., `real_part`), these definitions are now internal to different functions (see section 1.1.8), so there is no name conflict.

The complex-arithmetic selectors access the table by means of a general ‘operation’ function called `apply_generic`, which applies a generic operation to some arguments. The function `apply_generic` looks in the table under the name of the operation and the types of the argu-

⁴⁰We use the list `list("rectangular")` rather than the string "rectangular" to allow for the possibility of operations with multiple arguments, not all of the same type.

⁴¹The type the constructors are installed under needn't be a list because a constructor is always used to make an object of one particular type.

ments and applies the resulting function if one is present:⁴²

```
function apply_generic(op, args) {
  const type_tags = map(type_tag, args);
  const fun = get(op, type_tags);
  return fun !== undefined
    ? apply(fun, map(contents, args))
    : Error("No method for these types in apply_generic",
            list(op, type_tags));
}
```

Using `apply_generic`, we can define our generic selectors as follows:

```
function real_part(z) {
  return apply_generic("real_part", list(z));
}
function imag_part(z) {
  return apply_generic("imag_part", list(z));
}
function magnitude(z) {
  return apply_generic("magnitude", list(z));
}
function angle(z) {
  return apply_generic("angle", list(z));
}
```

Observe that these do not change at all if a new representation is added to the system.

We can also extract from the table the constructors to be used by the programs external to the packages in making complex numbers from real and imaginary parts and from magnitudes and angles. As in section 2.4.2, we construct rectangular numbers whenever we have real and imaginary parts, and polar numbers whenever we have magnitudes and angles:

```
function make_from_real_imag(x, y) {
  return get("make_from_real_imag", "rectangular")(x, y);
}
function make_from_mag_ang(r, a) {
  return get("make_from_mag_ang", "polar")(r, a);
}
```

Exercise 2.73

⁴²In `apply_generic`, `op` has as its value the first argument to `apply_generic` and `args` has as its value a list of the remaining arguments. The function `apply_generic` also uses the primitive function `apply`, which takes two arguments, a function and a list. The function `apply` applies the function, using the elements in the list as arguments. For example,

```
apply(sum_of_squares, list(1, 3))
returns 10.
```

Section 2.3.2 described a program that performs symbolic differentiation:

```
function deriv(exp, variable) {
  return is_number(exp)
    ? 0
    : is_variable(exp)
      ? (is_same_variable(exp, variable)) ? 1 : 0
    : is_sum(exp)
      ? make_sum(deriv(addend(exp), variable),
                  deriv(augend(exp), variable))
    : is_product(exp)
      ? make_sum(make_product(multiplier(exp),
                               deriv(multiplicand(exp),
                                     variable)),
                  make_product(deriv(multiplier(exp),
                                    variable),
                               multiplicand(exp)))
    // more rules can be added here
    : Error("unknown expression type in deriv",
           exp);
}

deriv(list("*", list("*", "x", "y"), list("+", "x", 4)), "x");
// [ "+",
//   [[ "*", [ ["*", ["x", ["y", null]]],,
//             [ ["+", [1, [0, null]], null]]],
//   [ ["*", [ [ "+",
//             [ ["*", [ "x", [0, null]]],,
//               [ ["*", [1, ["y", null]], null]]],,
//             [ ["+", [ "x", [4, null]], null] ] ],
//   null ]]
```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the ‘type tag’ of the datum is the algebraic operator symbol (such as +) and the operation being performed is `deriv`. We can transform this program into data-directed style by rewriting the basic derivative function as

```
function deriv(exp, variable) {
  return is_number(exp)
    ? 0
    : is_variable(exp)
      ? (is_same_variable(exp, variable)) ? 1 : 0
    : get("deriv",
          operator(exp))(operands(exp), variable);
}

function operator(exp) {
  return head(exp);
```

```

}
function operands(exp) {
    return tail(exp);
}

```

- a. Explain what was done above. Why can't we assimilate the predicates `is_number` and `is_same_variable` into the data-directed dispatch?
- b. Write the functions for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.
- c. Choose any additional differentiation rule that you like, such as the one for exponents (exercise 2.56), and install it in this data-directed system.
- d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the functions in the opposite way, so that the dispatch line in `deriv` looked like
`get(operator(exp), "deriv")(operands(exp), variable);`

What corresponding changes to the derivative system are required?

[Solution](#)

Exercise 2.74

Insatiable Enterprises, Inc., is a highly decentralized conglomerate company consisting of a large number of independent divisions located all over the world. The company's computer facilities have just been interconnected by means of a clever network-interfacing scheme that makes the entire network appear to any user to be a single computer. Insatiable's president, in her first attempt to exploit the ability of the network to extract administrative information from division files, is dismayed to discover that, although all the division files have been implemented as data structures in JavaScript, the particular data structure used varies from division to division. A meeting of division managers is hastily called to search for a strategy to integrate the files that will satisfy headquarters' needs while preserving the existing autonomy of the divisions. Show how such a strategy can be implemented with data-directed programming. As an example, suppose that each division's personnel records consist of a single file, which contains a set of records keyed on employees' names. The structure of the set varies from division to division. Furthermore, each employee's record is itself a set (structured differently from division to division) that contains information keyed under identifiers such as address and salary. In particular:

- a. Implement for headquarters a `get_record` function that retrieves a specified employee's record from a specified personnel file. The function should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?
- b. Implement for headquarters a `get_salary` function that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?
- c. Implement for headquarters a `find_employee_record` function. This should search all the divisions' files for the record of a given employee and return the record. Assume that this function takes as arguments an employee's name and a list of all the divisions' files.
- d. When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?

[Solution](#)

Message passing

The key idea of data-directed programming is to handle generic operations in programs by dealing explicitly with operation-and-type tables, such as the table in figure 2.22. The style of programming we used in section 2.4.2 organized the required dispatching on type by having each operation take care of its own dispatching. In effect, this decomposes the operation-and-type table into rows, with each generic operation function representing a row of the table.

An alternative implementation strategy is to decompose the table into columns and, instead of using ‘intelligent operations’ that dispatch on data types, to work with ‘intelligent data objects’ that dispatch on operation names. We can do this by arranging things so that a data object, such as a rectangular number, is represented as a function that takes as input the required operation name and performs the operation indicated. In such a discipline, `make_from_real_imag` could be written as

```
function make_from_real_imag(x, y) {
    function dispatch(op) {
        return op === "real_part"
            ? x
            : op === "imag_part"
            ? y
            : op === "magnitude"
            ? math_sqrt(square(x) + square(y))
            : op === "angle"
```

```

    ? math_atan(y, x)
    : Error("Unknown op in make_from_real_imag",
            op);
}
return dispatch;
}

```

The corresponding `apply_generic` function, which applies a generic operation to an argument, now simply feeds the operation's name to the data object and lets the object do the work:⁴³

```

function apply_generic(op, arg) {
    return head(arg)(op);
}

```

Note that the value returned by `make_from_real_imag` is a function—the internal `dispatch` function. This is the function that is invoked when `apply_generic` requests an operation to be performed.

This style of programming is called *message passing*. The name comes from the image that a data object is an entity that receives the requested operation name as a ‘message.’ We have already seen an example of message passing in section 2.1.3, where we saw how `pair`, `head`, and `tail` could be defined with no data objects but only functions. Here we see that message passing is not a mathematical trick but a useful technique for organizing systems with generic operations. In the remainder of this chapter we will continue to use data-directed programming, rather than message passing, to discuss generic arithmetic operations. In chapter 3 we will return to message passing, and we will see that it can be a powerful tool for structuring simulation programs.

Exercise 2.75

Implement the constructor `make_from_mag_ang` in message-passing style. This function should be analogous to the `make_from_real_imag` function given above.

[Solution](#)

Exercise 2.76

As a large system with generic operations evolves, new types of data objects or new operations may be needed. For each of the three strategies—generic operations with explicit dispatch, data-directed style, and message-passing-style—describe the changes that must be made to a system in order to add new types or new operations. Which organization would be most appropriate for a system in which new types must often be added? Which would be most appropriate for a system in which new operations must often be added?

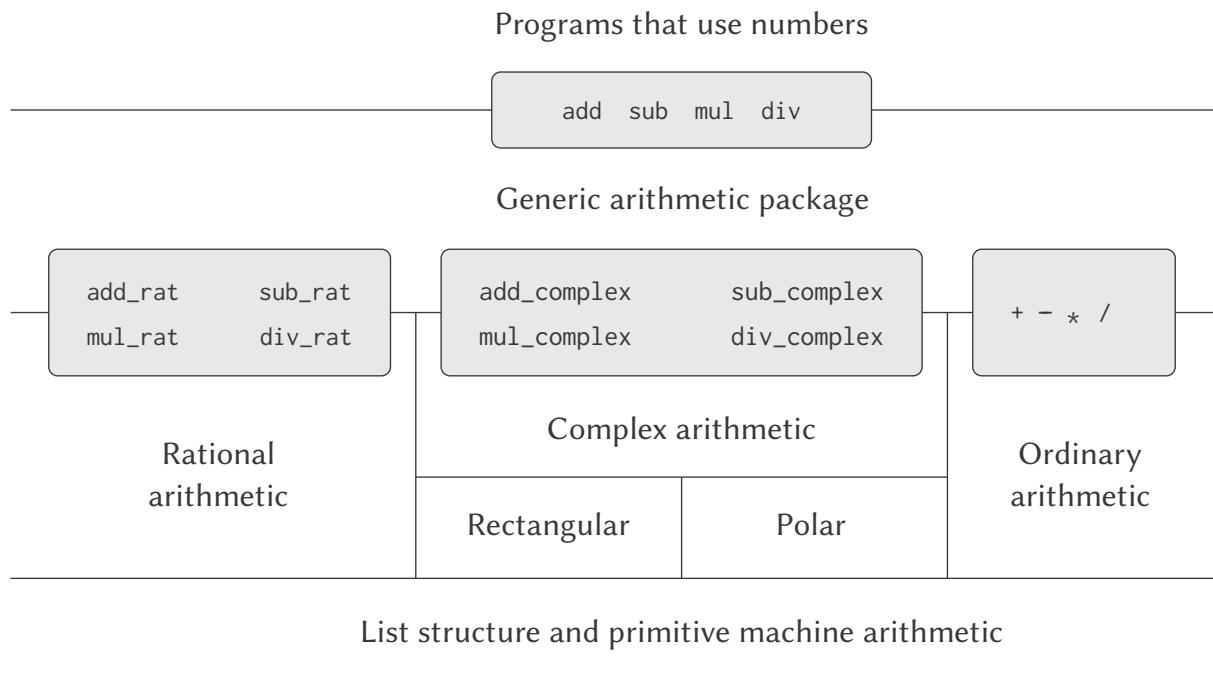
[Solution](#)

⁴³One limitation of this organization is it permits only generic functions of one argument.

2.5 Systems with Generic Operations

In the previous section, we saw how to design systems in which data objects can be represented in more than one way. The key idea is to link the code that specifies the data operations to the several representations by means of generic interface functions. Now we will see how to use this same idea not only to define operations that are generic over different representations but also to define operations that are generic over different kinds of arguments. We have already seen several different packages of arithmetic operations: the primitive arithmetic ($+$, $-$, $*$, $/$) built into our language, the rational-number arithmetic (`add_rat`, `sub_rat`, `mul_rat`, `div_rat`) of section 2.1.1, and the complex-number arithmetic that we implemented in section 2.4.3. We will now use data-directed techniques to construct a package of arithmetic operations that incorporates all the arithmetic packages we have already constructed.

Figure 2.23 shows the structure of the system we shall build. Notice the abstraction barriers. From the perspective of someone using ‘numbers,’ there is a single function `add` that operates on whatever numbers are supplied. The function `add` is part of a generic interface that allows the separate ordinary-arithmetic, rational-arithmetic, and complex-arithmetic packages to be accessed uniformly by programs that use numbers. Any individual arithmetic package (such as the complex package) may itself be accessed through generic functions (such as `add_complex`) that combine packages designed for different representations (such as rectangular and polar). Moreover, the structure of the system is additive, so that one can design the individual arithmetic packages separately and combine them to produce a generic arithmetic system.



List structure and primitive machine arithmetic

Figure 2.23: Generic arithmetic system.

2.5.1 Generic Arithmetic Operations

The task of designing generic arithmetic operations is analogous to that of designing the generic complex-number operations. We would like, for instance, to have a generic addition function add that acts like ordinary primitive addition + on ordinary numbers, like add_rat on rational numbers, and like add_complex on complex numbers. We can implement add, and the other generic arithmetic operations, by following the same strategy we used in section 2.4.3 to implement the generic selectors for complex numbers. We will attach a type tag to each kind of number and cause the generic function to dispatch to an appropriate package according to the data type of its arguments.

The generic arithmetic functions are defined as follows:

```

function add(x, y) {
    return apply_generic("add", list(x, y));
}

function sub(x, y) {
    return apply_generic("sub", list(x, y));
}

function mul(x, y) {
    return apply_generic("mul", list(x, y));
}

function div(x, y) {
    return apply_generic("div", list(x, y));
}

```

We begin by installing a package for handling *ordinary* numbers, that is, the primitive numbers

of our language. We will tag these with the symbol `javascript_number`. The arithmetic operations in this package are the primitive arithmetic functions (so there is no need to define extra functions to handle the untagged numbers). Since these operations each take two arguments, they are installed in the table keyed by the list `list("javascript_number", "javascript_number")`:

```
function install_javascript_number_package() {
    function tag(x) {
        return attach_tag("javascript_number", x);
    }
    put("add", list("javascript_number", "javascript_number"),
        (x, y) => tag(x + y));
    put("sub", list("javascript_number", "javascript_number"),
        (x, y) => tag(x - y));
    put("mul", list("javascript_number", "javascript_number"),
        (x, y) => tag(x * y));
    put("div", list("javascript_number", "javascript_number"),
        (x, y) => tag(x / y));
    put("make", "javascript_number",
        x => tag(x));
    return "done";
}
```

Users of the JavaScript-number package will create (tagged) ordinary numbers by means of the function:

```
function make_javascript_number(n) {
    return get("make", "javascript_number")(n);
}
```

Now that the framework of the generic arithmetic system is in place, we can readily include new kinds of numbers. Here is a package that performs rational arithmetic. Notice that, as a benefit of additivity, we can use without modification the rational-number code from section 2.1.1 as the internal functions in the package:

```
function install_rational_package() {
    // internal functions
    function numer(x) {
        return head(x);
    }
    function denom(x) {
        return tail(x);
    }
    function make_rat(n, d) {
        let g = gcd(n, d);
        return pair(n / g, d / g);
    }
    function add_rat(x, y) {
```

```

        return make_rat(numer(x) * denom(y) +
                        numer(y) * denom(x),
                        denom(x) * denom(y));
    }
function sub_rat(x, y) {
    return make_rat(numer(x) * denom(y) -
                    numer(y) * denom(x),
                    denom(x) * denom(y));
}
function mul_rat(x, y) {
    return make_rat(numer(x) * numer(y),
                    denom(x) * denom(y));
}
function div_rat(x, y) {
    return make_rat(numer(x) * denom(y),
                    denom(x) * numer(y));
}
// interface to rest of the system
function tag(x) {
    return attach_tag("rational", x);
}
put("make", "rational", make_rational);
put("add", list("rational", "rational"), add_rational);
put("sub", list("rational", "rational"), sub_rational);
put("mul", list("rational", "rational"), mul_rational);
put("div", list("rational", "rational"), div_rational);
}

function make_rational(n, d) {
    return (get("make", "rational"))(n, d);
}

```

We can install a similar package to handle complex numbers, using the tag "complex". In creating the package, we extract from the table the operations `make_from_real_imag` and `make_from_mag_ang` that were defined by the rectangular and polar packages. Additivity permits us to use, as the internal operations, the same `add_complex`, `sub_complex`, `mul_complex`, and `div_complex` functions from section 2.4.1.

```

function install_complex_package() {
    // imported functions from rectangular and polar packages
    function make_from_real_imag(x, y) {
        return get("make_from_real_imag", "rectangular")(x, y);
    }
    function make_from_mag_ang(r, a) {
        return get("make_from_mag_ang", "polar")(r, a);
    }

    // internal functions

```

```

function add_complex(z1, z2) {
    return make_from_real_imag(real_part(z1) +
                                real_part(z2),
                                imag_part(z1) +
                                imag_part(z2));
}

function sub_complex(z1, z2) {
    return make_from_real_imag(real_part(z1) -
                                real_part(z2),
                                imag_part(z1) -
                                imag_part(z2));
}

function mul_complex(z1, z2) {
    return make_from_mag_ang(magnitude(x) *
                                magnitude(z2),
                                angle(z1) +
                                angle(z2));
}

function div_complex(z1, z2) {
    return make_from_mag_ang(magnitude(x) /
                                magnitude(z2),
                                angle(z1) -
                                angle(z2));
}

// interface to rest of the system
function tag(z) {
    return attach_tag("complex", z);
}

put("add", list("complex", "complex"),
     (z1, z2) => tag(add_complex(z1, z2)));
put("sub", list("complex", "complex"),
     (z1, z2) => tag(sub_complex(z1, z2)));
put("mul", list("complex", "complex"),
     (z1, z2) => tag(mul_complex(z1, z2)));
put("div", list("complex", "complex"),
     (z1, z2) => tag(div_complex(z1, z2)));
put("make_from_real_imag", "complex",
     (x, y) => tag(make_from_real_imag(x, y)));
put("make_from_mag_ang", "complex",
     (r, a) => tag(make_from_mag_ang(r, a)));
return "done";
}

```

Programs outside the complex-number package can construct complex numbers either from real and imaginary parts or from magnitudes and angles. Notice how the underlying functions, originally defined in the rectangular and polar packages, are exported to the complex package,

and exported from there to the outside world.

```
function make_complex_from_real_imag(x, y){
    return get("make_from_real_imag", "complex")(x, y);
}
function make_complex_from_mag_ang(r, a){
    return get("make_from_mag_ang", "complex")(r, a);
}
```

What we have here is a two-level tag system. A typical complex number, such as $3 + 4i$ in rectangular form, would be represented as shown in Figure 2.24. The outer tag ("complex") is used to direct the number to the complex package. Once within the complex package, the next tag ("rectangular") is used to direct the number to the rectangular package. In a large and complicated system there might be many levels, each interfaced with the next by means of generic operations. As a data object is passed 'downward,' the outer tag that is used to direct it to the appropriate package is stripped off (by applying contents) and the next level of tag (if any) becomes visible to be used for further dispatching.

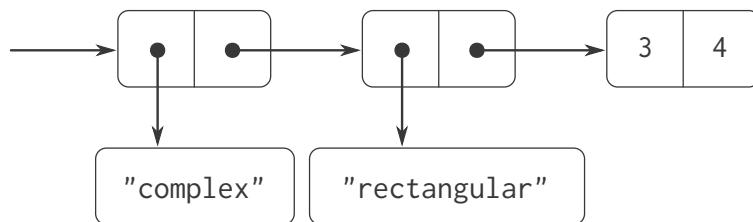


Figure 2.24: Representation of $3 + 4i$ in rectangular form.

In the above packages, we used `add_rat`, `add_complex`, and the other arithmetic functions exactly as originally written. Once these definitions are internal to different installation functions, however, they no longer need names that are distinct from each other: we could simply name them `add`, `sub`, `mul`, and `div` in both packages.

Exercise 2.77

Louis Reasoner tries to evaluate the expression `magnitude(z)` where `z` is the object shown in Figure 2.24. To his surprise, instead of the answer 5 he gets an error message from `apply_generic`, saying there is no method for the operation `magnitude` on the types `["complex", null]`. He shows this interaction to Alyssa P. Hacker, who says 'The problem is that the complex-number selectors were never defined for "complex" numbers, just for "polar" and "rectangular" numbers. All you have to do to make this work is add the following to the `complex` package:'

```
put("real_part", list("complex"), real_part);
put("imag_part", list("complex"), imag_part);
put("magnitude", list("complex"), magnitude);
put("angle", list("complex"), angle);
```

Describe in detail why this works. As an example, trace through all the functions called in evaluating the expression `magnitude(z)` where `z` is the object shown in Figure 2.24. In particular, how many times is `apply_generic` invoked? What function is dispatched to in each case?

Exercise 2.78

The internal functions in the `javascript_number` package are essentially nothing more than calls to the primitive functions `+`, `-`, etc. It was not possible to use the primitives of the language directly because our type-tag system requires that each data object have a type attached to it. In fact, however, all JavaScript implementations do have a type system, which they use internally. Primitive predicates such as `is_string` and `is_number` determine whether data objects have particular types. Modify the definitions of `type_tag`, `contents`, and `attach_tag` from section 2.4.2 so that our generic system takes advantage of JavaScript's internal type system. That is to say, the system should work as before except that ordinary numbers should be represented simply as JavaScript numbers rather than as pairs whose head is the string "`javascript_number`".

Exercise 2.79

Define a generic equality predicate `is_equ` that tests the equality of two numbers, and install it in the generic arithmetic package. This operation should work for ordinary numbers, rational numbers, and complex numbers.

Exercise 2.80

Define a generic predicate `is_equal_to_zero` that tests if its argument is zero, and install it in the generic arithmetic package. This operation should work for ordinary numbers, rational numbers, and complex numbers.

2.5.2 Combining Data of Different Types

We have seen how to define a unified arithmetic system that encompasses ordinary numbers, complex numbers, rational numbers, and any other type of number we might decide to invent, but we have ignored an important issue. The operations we have defined so far treat the different data types as being completely independent. Thus, there are separate packages for adding, say, two ordinary numbers, or two complex numbers. What we have not yet considered is the fact that it is meaningful to define operations that cross the type boundaries, such as the addition of a complex number to an ordinary number. We have gone to great pains to

introduce barriers between parts of our programs so that they can be developed and understood separately. We would like to introduce the cross-type operations in some carefully controlled way, so that we can support them without seriously violating our module boundaries.

One way to handle cross-type operations is to design a different function for each possible combination of types for which the operation is valid. For example, we could extend the complex-number package so that it provides a function for adding complex numbers to ordinary numbers and installs this in the table using the tag `list("complex", "javascript_number")`:⁴⁴

```
// to be included in the complex package
function add_complex_to_javascript_num(z, x) {
    return make_from_real_imag(real_part(z) + x,
                                imag_part(z));
}
put("add", list("complex", "javascript_number"),
    (z, x) => tag(add_complex_to_javascript_num(z, x)));
```

This technique works, but it is cumbersome. With such a system, the cost of introducing a new type is not just the construction of the package of functions for that type but also the construction and installation of the functions that implement the cross-type operations. This can easily be much more code than is needed to define the operations on the type itself. The method also undermines our ability to combine separate packages additively, or least to limit the extent to which the implementors of the individual packages need to take account of other packages. For instance, in the example above, it seems reasonable that handling mixed operations on complex numbers and ordinary numbers should be the responsibility of the complex-number package. Combining rational numbers and complex numbers, however, might be done by the complex package, by the rational package, or by some third package that uses operations extracted from these two packages. Formulating coherent policies on the division of responsibility among packages can be an overwhelming task in designing systems with many packages and many cross-type operations.

Coercion

In the general situation of completely unrelated operations acting on completely unrelated types, implementing explicit cross-type operations, cumbersome though it may be, is the best that one can hope for. Fortunately, we can usually do better by taking advantage of additional structure that may be latent in our type system. Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type. This process is called *coercion*. For example, if we are asked to arithmetically combine an ordinary number with a complex number, we can view the ordinary

⁴⁴We also have to supply an almost identical function to handle the types `list("javascript_number", "complex")`.

number as a complex number whose imaginary part is zero. This transforms the problem to that of combining two complex numbers, which can be handled in the ordinary way by the complex-arithmetic package.

In general, we can implement this idea by designing coercion functions that transform an object of one type into an equivalent object of another type. Here is a typical coercion function, which transforms a given ordinary number to a complex number with that real part and zero imaginary part:

```
function javascript_number_to_complex(n) {
    return make_complex_from_real_imag(contents(n), 0);
}
```

We install these coercion functions in a special coercion table, indexed under the names of the two types:

```
put_coercion("javascript_number",
              "complex",
              javascript_number_to_complex);
```

(We assume that there are `put_coercion` and `get_coercion` functions available for manipulating this table.) Generally some of the slots in the table will be empty, because it is not generally possible to coerce an arbitrary data object of each type into all other types. For example, there is no way to coerce an arbitrary complex number to an ordinary number, so there will be no general `complex_to_javascript_number` function included in the table.

Once the coercion table has been set up, we can handle coercion in a uniform manner by modifying the `apply_generic` function of section 2.4.3. When asked to apply an operation, we first check whether the operation is defined for the arguments' types, just as before. If so, we dispatch to the function found in the operation-and-type table. Otherwise, we try coercion. For simplicity, we consider only the case where there are two arguments.⁴⁵ We check the coercion table to see if objects of the first type can be coerced to the second type. If so, we coerce the first argument and try the operation again. If objects of the first type cannot in general be coerced to the second type, we try the coercion the other way around to see if there is a way to coerce the second argument to the type of the first argument. Finally, if there is no known way to coerce either type to the other type, we give up. Here is the function:

```
function apply_generic(op, args) {
    const type_tags = map(type_tag, args);
    const fun = get(op, type_tags);
    if (fun !== false) {
        return fun(map(contents, args));
    } else {
        if (length(args) === 2) {
            const type1 = head(type_tags);
```

⁴⁵See exercise 2.82 for generalizations.

Hierarchies of types

The coercion scheme presented above relied on the existence of natural relations between pairs of types. Often there is more ‘global’ structure in how the different types relate to each other. For instance, suppose we are building a generic arithmetic system to handle integers, rational numbers, real numbers, and complex numbers. In such a system, it is quite natural to regard an integer as a special kind of rational number, which is in turn a special kind of real number, which is in turn a special kind of complex number. What we actually have is a so-called *hierarchy of types*, in which, for example, integers are a *subtype* of rational numbers (i.e., any operation that can be applied to a rational number can automatically be applied to an integer). Conversely, we say that rational numbers form a *supertype* of integers. The particular hierarchy we have here is of a very simple kind, in which each type has at most one supertype and at most one subtype. Such a structure, called a *tower*, is illustrated in Figure 2.25.

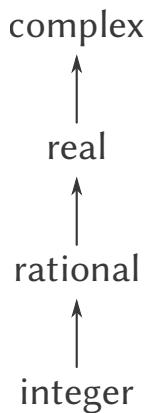


Figure 2.25: A tower of types.

If we have a tower structure, then we can greatly simplify the problem of adding a new type to the hierarchy, for we need only specify how the new type is embedded in the next supertype above it and how it is the supertype of the type below it. For example, if we want to add an integer to a complex number, we need not explicitly define a special coercion function `integer_to_complex`. Instead, we define how an integer can be transformed into a rational number, how a rational number is transformed into a real number, and how a real number is transformed into a complex number. We then allow the system to transform the integer into a complex number through these steps and then add the two complex numbers.

We can redesign our `apply_generic` function in the following way: For each type, we need to supply a `raise` function, which ‘raises’ objects of that type one level in the tower. Then when the system is required to operate on objects of different types it can successively raise the lower types until all the objects are at the same level in the tower. (Exercises 2.83 and 2.84 concern the details of implementing such a strategy.)

Another advantage of a tower is that we can easily implement the notion that every type

'inherits' all operations defined on a supertype. For instance, if we do not supply a special function for finding the real part of an integer, we should nevertheless expect that `real_part` will be defined for integers by virtue of the fact that integers are a subtype of complex numbers. In a tower, we can arrange for this to happen in a uniform way by modifying `apply_generic`. If the required operation is not directly defined for the type of the object given, we raise the object to its supertype and try again. We thus crawl up the tower, transforming our argument as we go, until we either find a level at which the desired operation can be performed or hit the top (in which case we give up).

Yet another advantage of a tower over a more general hierarchy is that it gives us a simple way to 'lower' a data object to the simplest representation. For example, if we add $2 + 3i$ to $4 - 3i$, it would be nice to obtain the answer as the integer 6 rather than as the complex number $6 + 0i$. Exercise 2.85 discusses a way to implement such a lowering operation. (The trick is that we need a general way to distinguish those objects that can be lowered, such as $6 + 0i$, from those that cannot, such as $6 + 2i$.)

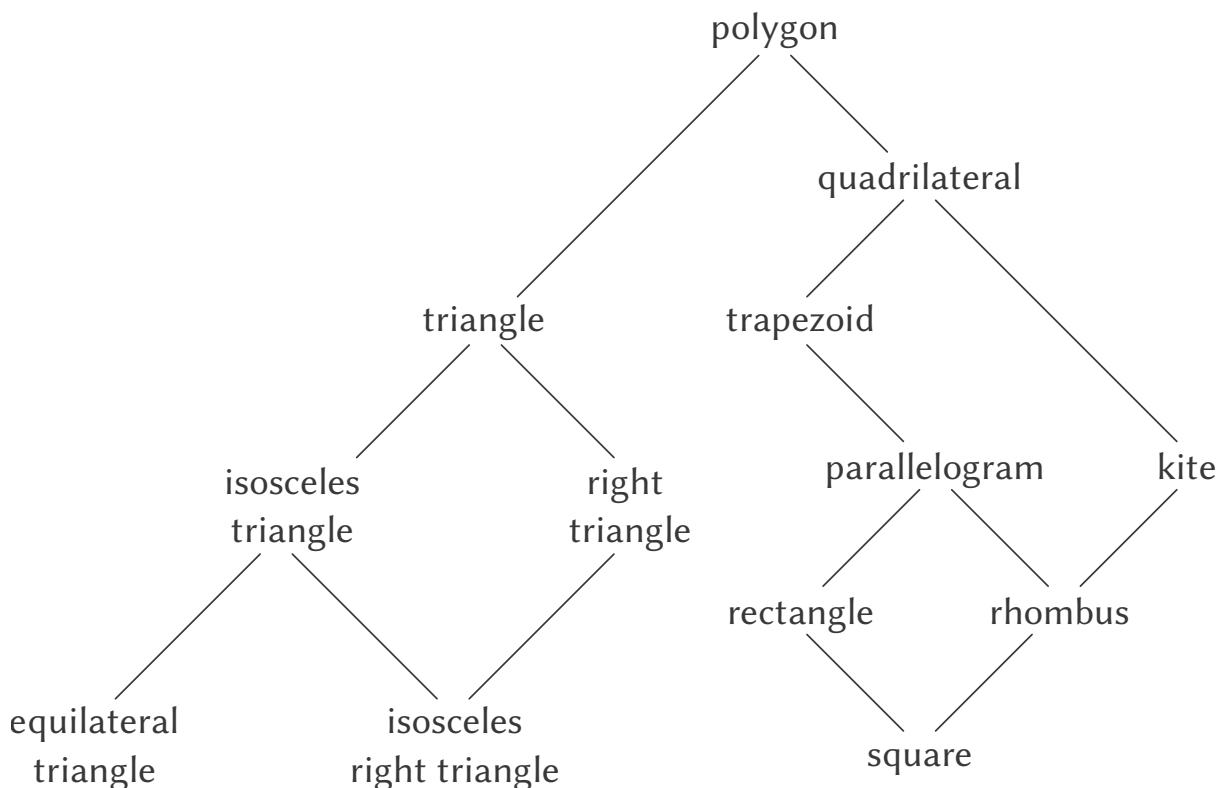


Figure 2.26: Relations among types of geometric figures.

Inadequacies of hierarchies

If the data types in our system can be naturally arranged in a tower, this greatly simplifies the problems of dealing with generic operations on different types, as we have seen. Unfortunately, this is usually not the case. Figure 2.26 illustrates a more complex arrangement of mixed types, this one showing relations among different types of geometric figures. We see that, in general, a type may have more than one subtype. Triangles and quadrilaterals, for instance, are both subtypes of polygons. In addition, a type may have more than one supertype. For example, an isosceles right triangle may be regarded either as an isosceles triangle or as a right triangle. This multiple-supertypes issue is particularly thorny, since it means that there is no unique way to ‘raise’ a type in the hierarchy. Finding the ‘correct’ supertype in which to apply an operation to an object may involve considerable searching through the entire type network on the part of a function such as `apply_generic`. Since there generally are multiple subtypes for a type, there is a similar problem in coercing a value ‘down’ the type hierarchy. Dealing with large numbers of interrelated types while still preserving modularity in the design of large systems is very difficult, and is an area of much current research.⁴⁷

Exercise 2.81

Louis Reasoner has noticed that `apply_generic` may try to coerce the arguments to each other’s type even if they already have the same type. Therefore, he reasons, we need to put functions in the coercion table to ‘coerce’ arguments of each type to their own type. For example, in addition to the `javascript_number_to_complex` coercion shown above, he would do:

```
function javascript_number_to_javascript_number(n) {
    return n;
}
function complex_number_to_complex_number(n) {
    return n;
}
put_coercion("javascript_number", "javascript_number",
    javascript_number_to_javascript_number);
```

⁴⁷This statement, which also appears in the first edition of this book, is just as true now as it was when we wrote it twelve years ago. Developing a useful, general framework for expressing the relations among different types of entities (what philosophers call ‘ontology’) seems intractably difficult. The main difference between the confusion that existed ten years ago and the confusion that exists now is that now a variety of inadequate ontological theories have been embodied in a plethora of correspondingly inadequate programming languages. For example, much of the complexity of object-oriented programming languages—and the subtle and confusing differences among contemporary object-oriented languages—centers on the treatment of generic operations on interrelated types. Our own discussion of computational objects in chapter 3 avoids these issues entirely. Readers familiar with object-oriented programming will notice that we have much to say in chapter 3 about local state, but we do not even mention ‘classes’ or ‘inheritance.’ In fact, we suspect that these problems cannot be adequately addressed in terms of computer-language design alone, without also drawing on work in knowledge representation and automated reasoning.

```
put_coercion("complex_number", "complex_number",
    complex_number_to_complex_number);
```

- a. With Louis's coercion functions installed, what happens if `apply_generic` is called with two arguments of type "javascript_number" or two arguments of type "complex" for an operation that is not found in the table for those types? For example, assume that we've defined a generic exponentiation operation:

```
function exp(x, y) {
    return apply_generic("exp", list(x, y));
}
```

and have put a function for exponentiation in the JavaScript-number package but not in any other package:

```
// following added to JavaScript-number package
put("exp", list("javascript_number", "javascript_number"),
    (x, y) => tag(math_exp(x, y))); // primitive math_exp
```

What happens if we call `exp` with two complex numbers as arguments?

- b. Is Louis correct that something had to be done about coercion with arguments of the same type, or does `apply_generic` work correctly as is?
- c. Modify `apply_generic` so that it doesn't try coercion if the two arguments have the same type.

Exercise 2.82

Show how to generalize `apply_generic` to handle coercion in the general case of multiple arguments. One strategy is to attempt to coerce all the arguments to the type of the first argument, then to the type of the second argument, and so on. Give an example of a situation where this strategy (and likewise the two-argument version given above) is not sufficiently general. (Hint: Consider the case where there are some suitable mixed-type operations present in the table that will not be tried.)

Exercise 2.83

Suppose you are designing a generic arithmetic system for dealing with the tower of types shown in Figure 2.25: integer, rational, real, complex. For each type (except complex), design a function that raises objects of that type one level in the tower. Show how to install a generic `raise` operation that will work for each type (except complex).

Exercise 2.84

Using the `raise` operation of exercise 2.83, modify the `apply_generic` function so that it coerces its arguments to have the same type by the method of successive raising, as discussed in this section. You will need to devise a way to test which of two types is higher in the tower. Do this in a manner that is ‘compatible’ with the rest of the system and will not lead to problems in adding new levels to the tower.

Exercise 2.85

This section mentioned a method for ‘simplifying’ a data object by lowering it in the tower of types as far as possible. Design a function `drop` that accomplishes this for the tower described in exercise 2.83. The key is to decide, in some general way, whether an object can be lowered. For example, the complex number $1.5 + 0i$ can be lowered as far as “real”, the complex number $1 + 0i$ can be lowered as far as “integer”, and the complex number $2 + 3i$ cannot be lowered at all. Here is a plan for determining whether an object can be lowered: Begin by defining a generic operation project that ‘pushes’ an object down in the tower. For example, projecting a complex number would involve throwing away the imaginary part. Then a number can be dropped if, when we project it and raise the result back to the type we started with, we end up with something equal to what we started with. Show how to implement this idea in detail, by writing a `drop` function that drops an object as far as possible. You will need to design the various projection operations⁴⁸ and install `project` as a generic operation in the system. You will also need to make use of a generic equality predicate, such as described in exercise 2.79. Finally, use `drop` to rewrite `apply_generic` from exercise 2.84 so that it ‘simplifies’ its answers.

Exercise 2.86

Suppose we want to handle complex numbers whose real parts, imaginary parts, magnitudes, and angles can be either ordinary numbers, rational numbers, or other numbers we might wish to add to the system. Describe and implement the changes to the system needed to accommodate this. You will have to define operations such as sine and cosine that are generic over ordinary numbers and rational numbers.

⁴⁸A real number can be projected to an integer using the `round` primitive, which returns the closest integer to its argument.

2.5.3 Example: Symbolic Algebra

The manipulation of symbolic algebraic expressions is a complex process that illustrates many of the hardest problems that occur in the design of large-scale systems. An algebraic expression, in general, can be viewed as a hierarchical structure, a tree of operators applied to operands. We can construct algebraic expressions by starting with a set of primitive objects, such as constants and variables, and combining these by means of algebraic operators, such as addition and multiplication. As in other languages, we form abstractions that enable us to refer to compound objects in simple terms. Typical abstractions in symbolic algebra are ideas such as linear combination, polynomial, rational function, or trigonometric function. We can regard these as compound ‘types,’ which are often useful for directing the processing of expressions. For example, we could describe the expression

$$x^2 \sin(y^2 + 1) + x \cos 2y + \cos(y^3 - 2y^2)$$

as a polynomial in x with coefficients that are trigonometric functions of polynomials in y whose coefficients are integers.

We will not attempt to develop a complete algebraic-manipulation system here. Such systems are exceedingly complex programs, embodying deep algebraic knowledge and elegant algorithms. What we will do is look at a simple but important part of algebraic manipulation: the arithmetic of polynomials. We will illustrate the kinds of decisions the designer of such a system faces, and how to apply the ideas of abstract data and generic operations to help organize this effort.

Arithmetic on polynomials

Our first task in designing a system for performing arithmetic on polynomials is to decide just what a polynomial is. Polynomials are normally defined relative to certain variables (the *indeterminates* of the polynomial). For simplicity, we will restrict ourselves to polynomials having just one indeterminate (*univariate polynomials*).⁴⁹ We will define a polynomial to be a sum of terms, each of which is either a coefficient, a power of the indeterminate, or a product of a coefficient and a power of the indeterminate. A coefficient is defined as an algebraic expression that is not dependent upon the indeterminate of the polynomial. For example,

$$5x^2 + 3x + 7$$

⁴⁹On the other hand, we will allow polynomials whose coefficients are themselves polynomials in other variables. This will give us essentially the same representational power as a full multivariate system, although it does lead to coercion problems, as discussed below.

is a simple polynomial in x , and

$$(y^2 + 1)x^3 + (2y)x + 1$$

is a polynomial in x whose coefficients are polynomials in y .

Already we are skirting some thorny issues. Is the first of these polynomials the same as the polynomial $5y^2 + 3y + 7$, or not? A reasonable answer might be ‘yes, if we are considering a polynomial purely as a mathematical function, but no, if we are considering a polynomial to be a syntactic form.’ The second polynomial is algebraically equivalent to a polynomial in y whose coefficients are polynomials in x . Should our system recognize this, or not? Furthermore, there are other ways to represent a polynomial—for example, as a product of factors, or (for a univariate polynomial) as the set of roots, or as a listing of the values of the polynomial at a specified set of points.⁵⁰ We can finesse these questions by deciding that in our algebraic-manipulation system a ‘polynomial’ will be a particular syntactic form, not its underlying mathematical meaning.

Now we must consider how to go about doing arithmetic on polynomials. In this simple system, we will consider only addition and multiplication. Moreover, we will insist that two polynomials to be combined must have the same indeterminate.

We will approach the design of our system by following the familiar discipline of data abstraction. We will represent polynomials using a data structure called a *poly*, which consists of a variable and a collection of terms. We assume that we have selectors `variable` and `term_list` that extract those parts from a *poly* and a constructor `make_poly` that assembles a *poly* from a given variable and a term list. A variable will be just a symbol, so we can use the `is_same_variable` function of section 2.3.2 to compare variables. The following functions define addition and multiplication of *polys*:

```
function add_poly(p1, p2) {
    return is_same_variable(variable(p1), variable(p2))
        ? make_poly(variable(p1),
                    add_terms(term_list(p1),
                               term_list(p2)))
        : Error("Polys not in same var in add_poly",
               list(p1, p2));
}
function mul_poly(p1, p2) {
    return is_same_variable(variable(p1), variable(p2))
        ? make_poly(variable(p1),
```

⁵⁰For univariate polynomials, giving the value of a polynomial at a given set of points can be a particularly good representation. This makes polynomial arithmetic extremely simple. To obtain, for example, the sum of two polynomials represented in this way, we need only add the values of the polynomials at corresponding points. To transform back to a more familiar representation, we can use the Lagrange interpolation formula, which shows how to recover the coefficients of a polynomial of degree n given the values of the polynomial at $n + 1$ points.

```

        mul_terms(term_list(p1),
                   term_list(p2)))
    : Error("Polys not in same var in mul_poly",
           list(p1, p2));
}

```

To incorporate polynomials into our generic arithmetic system, we need to supply them with type tags. We'll use the tag "polynomial", and install appropriate operations on tagged polynomials in the operation table. We'll embed all our code in an installation function for the polynomial package, similar to the ones in section 2.5.1:

```

function install_polynomial_package() {
    // internal functions
    // representation of poly
    function make_poly(variable, term_list) {
        return pair(variable, term_list);
    }
    function variable(p) { return head(p); }
    function term_list(p) { return tail(p); }
    // functions is_same_variable and
    // is_variable from section 2.3.2
    // representation of terms and term lists
    // functions adjoin_term ...coeff from text below

    function add_poly(p1, p2) {
        ...
    }
    // functions used by add_poly
    function mul_poly(p1, p2) {
        ...
    }
    // functions used by mul_poly
    // interface to rest of the system
    function tag(p) {
        return attach_tag("polynomial", p);
    }
    put("add", list("polynomial", "polynomial"),
        (p1, p2) => tag(add_poly(p1, p2)));
    put("mul", list("polynomial", "polynomial"),
        (p1, p2) => tag(mul_poly(p1, p2)));
    put("make", list("polynomial"),
        (variable, terms) =>
            tag(make_poly(variable, terms)));
    return "done";
}

```

Polynomial addition is performed termwise. Terms of the same order (i.e., with the same power of the indeterminate) must be combined. This is done by forming a new term of the same order

whose coefficient is the sum of the coefficients of the addends. Terms in one addend for which there are no terms of the same order in the other addend are simply accumulated into the sum polynomial being constructed.

In order to manipulate term lists, we will assume that we have a constructor `the_empty_termlist` that returns an empty term list and a constructor `adjoin_term` that adjoins a new term to a term list. We will also assume that we have a predicate `is_empty_termlist` that tells if a given term list is empty, a selector `first_term` that extracts the highest-order term from a term list, and a selector `rest_terms` that returns all but the highest-order term. To manipulate terms, we will suppose that we have a constructor `make_term` that constructs a term with given order and coefficient, and selectors `order` and `coeff` that return, respectively, the order and the coefficient of the term. These operations allow us to consider both terms and term lists as data abstractions, whose concrete representations we can worry about separately.

Here is the function that constructs the term list for the sum of two polynomials.⁵¹

```
function add_terms(L1, L2) {
  if (is_empty_termlist(L1)) {
    return L2;
  }
  else if (is_empty_termlist(L2)) {
    return L1;
  }
  else {
    const t1 = first_term(L1);
    const t2 = first_term(L2);
    if (order(t1) > order(t2)) {
      return adjoin_term(t1, add_terms(rest_terms(L1), L2));
    }
    else if (order(t1) < order(t2)) {
      return adjoin_term(t2, add_terms(L1, rest_terms(L2)));
    }
    else {
      return adjoin_term(make_term(order(t1),
                                    add(coeff(t1),
                                        coeff(t2))),
                        add_terms(rest_terms(L1),
                                rest_terms(L2)));
    }
  }
}
```

The most important point to note here is that we used the generic addition function `add` to add together the coefficients of the terms being combined. This has powerful consequences,

⁵¹This operation is very much like the `ordered_union_set` operation we developed in exercise 2.62. In fact, if we think of the terms of the polynomial as a set ordered according to the power of the indeterminate, then the program that produces the term list for a sum is almost identical to `union_set`.

as we will see below.

In order to multiply two term lists, we multiply each term of the first list by all the terms of the other list, repeatedly using `mul_term_by_all_terms`, which multiplies a given term by all terms in a given term list. The resulting term lists (one for each term of the first list) are accumulated into a sum. Multiplying two terms forms a term whose order is the sum of the orders of the factors and whose coefficient is the product of the coefficients of the factors:

```
function mul_terms(L1, L2) {
    return is_empty_termlist(L1)
        ? the_empty_termlist
        : add_terms(mul_term_by_all_terms(
                    first_term(L1), L2),
                    mul_terms(rest_terms(L1), L2));
}
function mul_term_by_all_terms(t1, L) {
    if (is_empty_termlist(L)) {
        return the_empty_termlist;
    } else {
        const t2 = first_term(L);
        return adjoin_term(make_term(order(t1) + order(t2),
                                    mul(coeff(t1), coeff(t2))),
                                    mul_term_by_all_terms(t1, rest_terms(L)));
    }
}
```

This is really all there is to polynomial addition and multiplication. Notice that, since we operate on terms using the generic functions `add` and `mul`, our polynomial package is automatically able to handle any type of coefficient that is known about by the generic arithmetic package. If we include a coercion mechanism such as one of those discussed in section 2.5.2, then we also are automatically able to handle operations on polynomials of different coefficient types, such as

$$\left[3x^2 + (2 + 3i)x + 7\right] \cdot \left[x^4 + \frac{2}{3}x^2 + (5 + 3i)\right]$$

Because we installed the polynomial addition and multiplication functions `add_poly` and `mul_poly` in the generic arithmetic system as the `add` and `mul` operations for type `polynomial`, our system is also automatically able to handle polynomial operations such as

$$[(y + 1)x^2 + (y^2 + 1)x + (y - 1)] \cdot [(y - 2)x + (y^3 + 7)]$$

The reason is that when the system tries to combine coefficients, it will dispatch through `add` and `mul`. Since the coefficients are themselves polynomials (in y), these will be combined using `add_poly` and `mul_poly`. The result is a kind of ‘data-directed recursion’ in which, for example, a call to `mul_poly` will result in recursive calls to `mul_poly` in order to multiply the

coefficients. If the coefficients of the coefficients were themselves polynomials (as might be used to represent polynomials in three variables), the data direction would ensure that the system would follow through another level of recursive calls, and so on through as many levels as the structure of the data dictates.⁵²

Representing term lists

Finally, we must confront the job of implementing a good representation for term lists. A term list is, in effect, a set of coefficients keyed by the order of the term. Hence, any of the methods for representing sets, as discussed in section 2.3.3, can be applied to this task. On the other hand, our functions `add_terms` and `mul_terms` always access term lists sequentially from highest to lowest order. Thus, we will use some kind of ordered list representation.

How should we structure the list that represents a term list? One consideration is the ‘density’ of the polynomials we intend to manipulate. A polynomial is said to be *dense* if it has nonzero coefficients in terms of most orders. If it has many zero terms it is said to be *sparse*. For example,

$$A : \quad x^5 + 2x^4 + 3x^2 - 2x - 5$$

is a dense polynomial, whereas

$$B : \quad x^{100} + 2x^2 + 1$$

is sparse.

The term lists of dense polynomials are most efficiently represented as lists of the coefficients. For example, *A* above would be nicely represented as `list(1, 2, 0, 3, -2, -5)`. The order of a term in this representation is the length of the sublist beginning with that term’s coefficient, decremented by 1.⁵³ This would be a terrible representation for a sparse polynomial such as *B*: There would be a giant list of zeros punctuated by a few lonely nonzero terms. A more reasonable representation of the term list of a sparse polynomial is as a list of the nonzero terms, where each term is a list containing the order of the term and the coefficient for that order. In such a scheme, polynomial *B* is efficiently represented as `list(list(100, 1), list(2, 2), list(0, 1))`. As most polynomial manipulations are performed on sparse polynomials, we will use this method. We will assume that term lists are represented as lists of terms, arranged from highest-

⁵²To make this work completely smoothly, we should also add to our generic arithmetic system the ability to coerce a ‘number’ to a polynomial by regarding it as a polynomial of degree zero whose coefficient is the number. This is necessary if we are going to perform operations such as

$$[x^2 + (y + 1)x + 5] + [x^2 + 2x + 1]$$

which requires adding the coefficient $y + 1$ to the coefficient 2.

⁵³In these polynomial examples, we assume that we have implemented the generic arithmetic system using the type mechanism suggested in exercise 2.78. Thus, coefficients that are ordinary numbers will be represented as the numbers themselves rather than as pairs whose head is the symbol `javascript_number`.

order to lowest-order term. Once we have made this decision, implementing the selectors and constructors for terms and term lists is straightforward:⁵⁴

```
function adjoin_term(term, term_list) {
  return is_equal_to_zero(coef(term))
    ? term_list
    : pair(term, term_list);
}
const the_empty_termlist = null;
function first_term(term_list) {
  return head(term_list);
}
function rest_terms(term_list) {
  return tail(term_list);
}
function is_empty_termlist(term_list) {
  return is_null(term_list);
}
function make_term(order, coeff) {
  return list(order, coeff);
}
function order(term) {
  return head(term);
}
function coeff(term) {
  return head(tail(term));
}
```

where `is_equal_to_zero` is as defined in exercise 2.80. (See also exercise 2.87 below.)

Users of the polynomial package will create (tagged) polynomials by means of the function:

```
function make_polynomial(variable, terms) {
  return get("make", "polynomial")(variable, terms);
}
```

Exercise 2.87

Install `is_equal_to_zero` for polynomials in the generic arithmetic package. This will allow `adjoin_term` to work for polynomials with coefficients that are themselves polynomials.

Exercise 2.88

⁵⁴Although we are assuming that term lists are ordered, we have implemented `adjoin_term` to simply pair the new term onto the existing term list. We can get away with this so long as we guarantee that the functions (such as `add_terms`) that use `adjoin_term` always call it with a higher-order term than appears in the list. If we did not want to make such a guarantee, we could have implemented `adjoin_term` to be similar to the `adjoin_set` constructor for the ordered-list representation of sets (exercise 2.61).

Extend the polynomial system to include subtraction of polynomials. (Hint: You may find it helpful to define a generic negation operation.)

Exercise 2.89

Define functions that implement the term-list representation described above as appropriate for dense polynomials.

Exercise 2.90

Suppose we want to have a polynomial system that is efficient for both sparse and dense polynomials. One way to do this is to allow both kinds of term-list representations in our system. The situation is analogous to the complex-number example of section 2.4, where we allowed both rectangular and polar representations. To do this we must distinguish different types of term lists and make the operations on term lists generic. Redesign the polynomial system to implement this generalization. This is a major effort, not a local change.

Exercise 2.91

A univariate polynomial can be divided by another one to produce a polynomial quotient and a polynomial remainder. For example,

$$\frac{x^5 - 1}{x^2 - 1} = x^3 + x, \text{ remainder } x - 1$$

Division can be performed via long division. That is, divide the highest-order term of the dividend by the highest-order term of the divisor. The result is the first term of the quotient. Next, multiply the result by the divisor, subtract that from the dividend, and produce the rest of the answer by recursively dividing the difference by the divisor. Stop when the order of the divisor exceeds the order of the dividend and declare the dividend to be the remainder. Also, if the dividend ever becomes zero, return zero as both quotient and remainder. We can design a `div_poly` function on the model of `add_poly` and `mul_poly`. The function checks to see if the two polys have the same variable. If so, `div_poly` strips off the variable and passes the problem to `div_terms`, which performs the division operation on term lists. `div_poly` finally reattaches the variable to the result supplied by `div_terms`. It is convenient to design `div_terms` to compute both the quotient and the remainder of a division. `div_terms` can take two term lists as arguments and return a list of the quotient term list and the remainder term list. Complete the following definition of `div_terms` by filling in the missing expressions. Use this to implement `div_poly`, which takes two polys as arguments and returns a list of the quotient and remainder polys.

```

function div_terms(L1, L2) {
  if (is_empty_termlist(L1)) {
    return list(the_empty_termlist, the_empty_termlist);
  } else {
    const t1 = first_term(L1);
    const t2 = first_term(L2);
    if (order(t2) > order(t1)) {
      return list(the_empty_termlist, L1);
    } else {
      const new_c = div(coeff(t1), coeff(t2));
      const new_o = order(t1) - order(t2);
      const rest_of_result = ...
        // compute rest_of_result recursively
      ...
      // form complete result
    }
  }
}

```

Hierarchies of types in symbolic algebra

Our polynomial system illustrates how objects of one type (polynomials) may in fact be complex objects that have objects of many different types as parts. This poses no real difficulty in defining generic operations. We need only install appropriate generic operations for performing the necessary manipulations of the parts of the compound types. In fact, we saw that polynomials form a kind of ‘recursive data abstraction,’ in that parts of a polynomial may themselves be polynomials. Our generic operations and our data-directed programming style can handle this complication without much trouble.

On the other hand, polynomial algebra is a system for which the data types cannot be naturally arranged in a tower. For instance, it is possible to have polynomials in x whose coefficients are polynomials in y . It is also possible to have polynomials in y whose coefficients are polynomials in x . Neither of these types is ‘above’ the other in any natural way, yet it is often necessary to add together elements from each set. There are several ways to do this. One possibility is to convert one polynomial to the type of the other by expanding and rearranging terms so that both polynomials have the same principal variable. One can impose a towerlike structure on this by ordering the variables and thus always converting any polynomial to a ‘canonical form’ with the highest-priority variable dominant and the lower-priority variables buried in the coefficients. This strategy works fairly well, except that the conversion may expand a polynomial unnecessarily, making it hard to read and perhaps less efficient to work with. The tower strategy is certainly not natural for this domain or for any domain where the user can invent new types dynamically using old types in various combining forms, such as trigonometric functions, power series, and integrals.

It should not be surprising that controlling coercion is a serious problem in the design of large-scale algebraic-manipulation systems. Much of the complexity of such systems is concerned with relationships among diverse types. Indeed, it is fair to say that we do not yet completely understand coercion. In fact, we do not yet completely understand the concept of a data type. Nevertheless, what we know provides us with powerful structuring and modularity principles to support the design of large systems.

Exercise 2.92

By imposing an ordering on variables, extend the polynomial package so that addition and multiplication of polynomials works for polynomials in different variables. (This is not easy!)

Extended exercise: Rational functions

We can extend our generic arithmetic system to include *rational functions*. These are ‘fractions’ whose numerator and denominator are polynomials, such as

$$\frac{x+1}{x^3-1}$$

The system should be able to add, subtract, multiply, and divide rational functions, and to perform such computations as

$$\frac{x+1}{x^3-1} + \frac{x}{x^2-1} = \frac{x^3+2x^2+3x+1}{x^4+x^3-x-1}$$

(Here the sum has been simplified by removing common factors. Ordinary ‘cross multiplication’ would have produced a fourth-degree polynomial over a fifth-degree polynomial.)

If we modify our rational-arithmetic package so that it uses generic operations, then it will do what we want, except for the problem of reducing fractions to lowest terms.

Exercise 2.93

Modify the rational-arithmetic package to use generic operations, but change `make_rat` so that it does not attempt to reduce fractions to lowest terms. Test your system by calling `make_rational` on two polynomials to produce a rational function

```
const p1 = make_polynomial("x", list(list(2, 1),
                                         list(0, 1)));
const p2 = make_polynomial("x", list(list(3, 1),
                                         list(0, 1)));
const rf = make_rational(p2, p1);
```

Now add `rf` to itself, using `add`. You will observe that this addition function does not reduce fractions to lowest terms.

We can reduce polynomial fractions to lowest terms using the same idea we used with integers: modifying `make_rat` to divide both the numerator and the denominator by their greatest common divisor. The notion of ‘greatest common divisor’ makes sense for polynomials. In fact, we can compute the GCD of two polynomials using essentially the same Euclid’s Algorithm that works for integers.⁵⁵ The integer version is

```
function gcd(a, b) {
  return b === 0
    ? a
    : gcd(b, remainder(a, b));
}
```

Using this, we could make the obvious modification to define a GCD operation that works on term lists:

```
function gcd_terms(a, b) {
  return is_empty_termlist(b)
    ? a
    : gcd_terms(b, remainder_terms(a, b));
}
```

where `remainder_terms` picks out the remainder component of the list returned by the term-list division operation `div_terms` that was implemented in exercise 2.91.

Exercise 2.94

Using `div_terms`, implement the function `remainder_terms` and use this to define `gcd_terms` as above. Now write a function `gcd_poly` that computes the polynomial GCD of two polys. (The function should signal an error if the two polys are not in the same variable.) Install in the system a generic operation `greatest_common_divisor` that reduces to `gcd_poly` for polynomials and to ordinary `gcd` for ordinary numbers. As a test, try

```
const p1 = make_polynomial("x", list(make_term(4, 1),
                                         make_term(3, -1),
                                         make_term(2, -2),
                                         make_term(1, 2)));
const p2 = make_polynomial("x", list(make_term(3, 1),
```

⁵⁵The fact that Euclid’s Algorithm works for polynomials is formalized in algebra by saying that polynomials form a kind of algebraic domain called a *Euclidean ring*. A Euclidean ring is a domain that admits addition, subtraction, and commutative multiplication, together with a way of assigning to each element x of the ring a positive integer ‘measure’ $m(x)$ with the properties that $m(xy) \geq m(x)$ for any nonzero x and y and that, given any x and y , there exists a q such that $y = qx + r$ and either $r = 0$ or $m(r) < m(x)$. From an abstract point of view, this is what is needed to prove that Euclid’s Algorithm works. For the domain of integers, the measure m of an integer is the absolute value of the integer itself. For the domain of polynomials, the measure of a polynomial is its degree.

```
make_term(1, -1)));
greatest_common_divisor(p1, p2);
```

and check your result by hand.

Exercise 2.95

Define P_1 , P_2 , and P_3 to be the polynomials

$$\begin{aligned}P_1 &: x^2 - 2x + 1 \\P_2 &: 11x^2 + 7 \\P_3 &: 13x + 5\end{aligned}$$

Now define Q_1 to be the product of P_1 and P_2 and Q_2 to be the product of P_1 and P_3 , and use `greatest_common_divisor` (exercise 2.94) to compute the GCD of Q_1 and Q_2 . Note that the answer is not the same as P_1 . This example introduces noninteger operations into the computation, causing difficulties with the GCD algorithm. To understand what is happening, try tracing `gcd_terms` while computing the GCD or try performing the division by hand.

We can solve the problem exhibited in exercise 2.95 if we use the following modification of the GCD algorithm (which really works only in the case of polynomials with integer coefficients). Before performing any polynomial division in the GCD computation, we multiply the dividend by an integer constant factor, chosen to guarantee that no fractions will arise during the division process. Our answer will thus differ from the actual GCD by an integer constant factor, but this does not matter in the case of reducing rational functions to lowest terms; the GCD will be used to divide both the numerator and denominator, so the integer constant factor will cancel out.

More precisely, if P and Q are polynomials, let O_1 be the order of P (i.e., the order of the largest term of P) and let O_2 be the order of Q . Let c be the leading coefficient of Q . Then it can be shown that, if we multiply P by the *integerizing factor* $c^{1+O_1-O_2}$, the resulting polynomial can be divided by Q by using the `div_terms` algorithm without introducing any fractions. The operation of multiplying the dividend by this constant and then dividing is sometimes called the *pseudodivision* of P by Q . The remainder of the division is called the *pseudoremainder*.

Exercise 2.96

- Implement the function `pseudoremainder_terms`, which is just like `remainder_terms` except that it multiplies the dividend by the integerizing factor described above before calling `div_terms`. Modify `gcd_terms` to use `pseudoremainder_terms`, and verify that

`greatest_common_divisor` now produces an answer with integer coefficients on the example in exercise 2.95.

- b. The GCD now has integer coefficients, but they are larger than those of P_1 . Modify `gcd_terms` so that it removes common factors from the coefficients of the answer by dividing all the coefficients by their (integer) greatest common divisor.

Thus, here is how to reduce a rational function to lowest terms:

- Compute the GCD of the numerator and denominator, using the version of `gcd_terms` from exercise 2.96.
- When you obtain the GCD, multiply both numerator and denominator by the same integerizing factor before dividing through by the GCD, so that division by the GCD will not introduce any noninteger coefficients. As the factor you can use the leading coefficient of the GCD raised to the power $1 + O_1 - O_2$, where O_2 is the order of the GCD and O_1 is the maximum of the orders of the numerator and denominator. This will ensure that dividing the numerator and denominator by the GCD will not introduce any fractions.
- The result of this operation will be a numerator and denominator with integer coefficients. The coefficients will normally be very large because of all of the integerizing factors, so the last step is to remove the redundant factors by computing the (integer) greatest common divisor of all the coefficients of the numerator and the denominator and dividing through by this factor.

Exercise 2.97

- a. Implement this algorithm as a function `reduce_terms` that takes two term lists `n` and `d` as arguments and returns a list `nn, dd`, which are `n` and `d` reduced to lowest terms via the algorithm given above. Also write a function `reduce_poly`, analogous to `add_poly`, that checks to see if the two polys have the same variable. If so, `reduce_poly` strips off the variable and passes the problem to `reduce_terms`, then reattaches the variable to the two term lists supplied by `reduce_terms`.
- b. Define a function analogous to `reduce_terms` that does what the original `make_rat` did for integers:

```
function reduce_integers(n, d) {
    const g = gcd(n, d);
    return list(n / g, d / g);
}
```

and define `reduce` as a generic operation that calls `apply_generic` to dispatch to either `reduce_poly` (for polynomial arguments) or `reduce_integers` (for `javascript_number` arguments). You can now easily make the rational-arithmetic package reduce fractions to lowest terms by having `make_rat` call `reduce` before combining the given numerator and denominator to form a rational number. The system now handles rational expressions in either integers or polynomials. To test your program, try the example at the beginning of this extended exercise:

```
const p1 = make_polynomial("x", list(make_term(1, 1),
                                      make_term(0, 1)));
const p2 = make_polynomial("x", list(make_term(3, 1),
                                      make_term(0, -1)));
const p3 = make_polynomial("x", list(make_term(1, 1)));
const p4 = make_polynomial("x", list(make_term(2, 1),
                                      make_term(0, -1)));

const rf1 = make_rational(p1, p2);
const rf2 = make_rational(p3, p4);

add(rf1, rf2);
```

See if you get the correct answer, correctly reduced to lowest terms.

The GCD computation is at the heart of any system that does operations on rational functions. The algorithm used above, although mathematically straightforward, is extremely slow. The slowness is due partly to the large number of division operations and partly to the enormous size of the intermediate coefficients generated by the pseudodivisions. One of the active areas in the development of algebraic-manipulation systems is the design of better algorithms for computing polynomial GCDs.⁵⁶

⁵⁶One extremely efficient and elegant method for computing polynomial GCDs was discovered by Richard Zippel (1979). The method is a probabilistic algorithm, as is the fast test for primality that we discussed in chapter 1. Zippel's book (1993) describes this method, together with other ways to compute polynomial GCDs.

Chapter 3

Modularity, Objects, and State

Μεταβάλλον – να παύεται (Even while it changes, it stands still.)

— Heraclitus

Plus Ça change, plus c'est la même chose.

— Alphonse Karr

The preceding chapters introduced the basic elements from which programs are made. We saw how primitive functions and primitive data are combined to construct compound entities, and we learned that abstraction is vital in helping us to cope with the complexity of large systems. But these tools are not sufficient for designing programs. Effective program synthesis also requires organizational principles that can guide us in formulating the overall design of a program. In particular, we need strategies to help us structure large systems so that they will be *modular*, that is, so that they can be divided ‘naturally’ into coherent parts that can be separately developed and maintained.

One powerful design strategy, which is particularly appropriate to the construction of programs for modeling physical systems, is to base the structure of our programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

To a large extent, then, the way we organize a large program is dictated by our perception of the system to be modeled. In this chapter we will investigate two prominent organizational strategies arising from two rather different ‘world views’ of the structure of systems. The first organizational strategy concentrates on *objects*, viewing a large system as a collection of distinct objects whose behaviors may change over time. An alternative organizational strategy concentrates on the *streams* of information that flow in the system, much as an electrical engineer views a signal-processing system.

Both the object-based approach and the stream-processing approach raise significant linguistic issues in programming. With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to abandon our old substitution model of computation (section 1.1.5) in favor of a more mechanistic but less theoretically tractable *environment model* of computation. The difficulties of dealing with objects, change, and identity are a fundamental consequence of the need to grapple with time in our computational models. These difficulties become even greater when we allow the possibility of concurrent execution of programs. The stream approach can be most fully exploited when we decouple simulated time in our model from the order of the events that take place in the computer during evaluation. We will accomplish this using a technique known as *delayed evaluation*.

3.1 Assignment and Local State

We ordinarily view the world as populated by independent objects, each of which has a state that changes over time. An object is said to ‘have state’ if its behavior is influenced by its history. A bank account, for example, has state in that the answer to the question ‘Can I withdraw \$100?’ depends upon the history of deposit and withdrawal transactions. We can characterize an object’s state by one or more *state variables*, which among them maintain enough information about history to determine the object’s current behavior. In a simple banking system, we could characterize the state of an account by a current balance rather than by remembering the entire history of account transactions.

In a system composed of many objects, the objects are rarely completely independent. Each may influence the states of others through interactions, which serve to couple the state variables of one object to those of other objects. Indeed, the view that a system is composed of separate objects is most useful when the state variables of the system can be grouped into closely coupled subsystems that are only loosely coupled to other subsystems.

This view of a system can be a powerful framework for organizing computational models of the system. For such a model to be modular, it should be decomposed into computational objects that model the actual objects in the system. Each computational object must have its own *local state variables* describing the actual object’s state. Since the states of objects in the

system being modeled change over time, the state variables of the corresponding computational objects must also change. If we choose to model the flow of time in the system by the elapsed time in the computer, then we must have a way to construct computational objects whose behaviors change as our programs run. In particular, if we wish to model state variables by ordinary symbolic names in the programming language, then the language must provide an *assignment operator* to enable us to change the value associated with a name.

3.1.1 Local State Variables

To illustrate what we mean by having a computational object with time-varying state, let us model the situation of withdrawing money from a bank account. We will do this using a function `withdraw`, which takes as argument an amount to be withdrawn. If there is enough money in the account to accommodate the withdrawal, then `withdraw` should return the balance remaining after the withdrawal. Otherwise, `withdraw` should return the message *Insufficient funds*. For example, if we begin with \$100 in the account, we should obtain the following sequence of responses using `withdraw`:

```
withdraw(25); // output: 75
withdraw(25); // output: 50
withdraw(60); // output: "Insufficient funds"
withdraw(15); // output: 35
```

Observe that the expression `withdraw(25)`, evaluated twice, yields different values. This is a new kind of behavior for a function. Until now, all our functions could be viewed as specifications for computing mathematical functions. A call to a function computed the value of the function applied to the given arguments, and two calls to the same function with the same arguments always produced the same result.¹

So far, all our names were *constants* as declared by the keyword `const`. Once declared, they did not change their value, as appropriate for constants. To implement functions like `withdraw`, we introduce a new kind of declaration—*variable declarations* using the keyword `let` instead of `const`. After declaring a variable `balance`, to indicate the balance of money in the account, we can define `withdraw` as a function that accesses `balance`. The `withdraw` function checks to see if `balance` is at least as large as the requested amount. If so, `withdraw` decrements `balance` by `amount` and returns the new value of `balance`. Otherwise, `withdraw` returns the *Insufficient funds* message. Here are the declarations of `balance` and `withdraw`:

¹Actually, this is not quite true. One exception was the random-number generator in section 1.2.6. Another exception involved the operation/type tables we introduced in section 2.4.3, where the values of two calls to get with the same arguments depended on intervening calls to put. On the other hand, until we introduce assignment, we have no way to create such functions ourselves.

```
let balance = 100;

function withdraw(amount) {
    if (balance >= amount) {
        balance = balance - amount;
        return balance;
    } else {
        return "Insufficient funds";
    }
}
```

Decrementing balance is accomplished by the statement

```
balance = balance - amount;
```

The syntax of such *assignment statements* is

```
name = new-value;
```

Here *name* is a symbol and *new-value* is any expression. The assignment changes *name* so that its value is the result obtained by evaluating *new-value*. In the case at hand, we are changing *balance* so that its new value will be the result of subtracting *amount* from the previous value of *balance*.²

The function *withdraw* also uses a *sequential composition* to cause two expressions to be evaluated in the case where the **if** test is true: first decrementing *balance* and then returning the value of *balance*. In general, executing the statement

```
stmt1 stmt2
```

causes the statements *stmt₁* and *stmt₂* to be evaluated in sequence.³

Although *withdraw* works as desired, the variable *balance* presents a problem. As specified above, *balance* is a name defined in the global environment and is freely accessible to be examined or modified by any function. It would be much better if we could somehow make *balance* internal to *withdraw*, so that *withdraw* would be the only function that could access

²Note that assignment statements look similar to and should not be confused with constant and variable declarations of the form

```
const name = value;
```

and

```
let name = value;
```

in which a newly declared *name* is associated with a *value*. Also similar in looks but not in meaning are expressions of the form

```
expression1 === expression2
```

which evaluate to true if *expression₁* evaluates to the same value as *expression₂* and to false, otherwise.

³We have already used sequential composition implicitly in our programs, because in JavaScript the body of a function can be a sequence of statements, not just a single **return** statement, as discussed in section 1.1.8.

balance directly and any other function could access balance only indirectly (through calls to withdraw). This would more accurately model the notion that balance is a local state variable used by withdraw to keep track of the state of the account.

We can make balance internal to withdraw by rewriting the definition as follows:

```
function make_withdraw() {
  let balance = 100;
  return amount => {
    if (balance >= amount) {
      balance = balance - amount;
      return balance;
    } else {
      return "insufficient funds";
    }
  };
}
const new_withdraw = make_withdraw();
```

What we have done here is use **let** to establish an environment with a local variable balance, bound to the initial value 100. Within this local environment, we use function definition⁴ to create a function that takes amount as an argument and behaves like our previous withdraw function. This function—returned as the result of evaluating the body of the make_withdraw function—behaves in precisely the same way as withdraw but whose variable balance is not accessible by any other function.⁵

Combining assignment statements with variable statements is the general programming technique we will use for constructing computational objects with local state. Unfortunately, using this technique raises a serious problem: When we first introduced functions, we also introduced the substitution model of evaluation (section 1.1.5) to provide an interpretation of what function application means. We said that applying a function should be interpreted as evaluating the body of the function with the formal parameters replaced by their values. The trouble is that, as soon as we introduce assignment into our language, substitution is no longer an adequate model of function application. (We will see why this is so in section 3.1.3.) As a consequence, we technically have at this point no way to understand why the new_withdraw function behaves as claimed above. In order to really understand a function such as new_withdraw, we will need to develop a new model of function application. In section 3.2 we will introduce such a model, together with an explanation of assignment statements and variable statements. First, however, we examine some variations on the theme established by make_withdraw.

⁴Blocks as bodies of function definition expressions were introduced in section 2.2.4.

⁵In programming-language jargon, the variable balance is said to be *encapsulated* within the new_withdraw function. Encapsulation reflects the general system-design principle known as the *hiding principle*: One can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a ‘need to know’.

The following function, `make_withdraw_with_balance`, abstracts the initial balance into a parameter. The formal parameter `balance` in `make_withdraw_with_balance` specifies the initial amount of money in the account.⁶

```
function make_withdraw_with_balance(balance) {
    return amount => {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "insufficient funds";
        }
    };
}
```

The function `make_withdraw_with_balance` can be used as follows to create two objects `w1` and `w2`:

```
const w1 = make_withdraw_with_balance(100);
const w2 = make_withdraw_with_balance(100);

w1(50); // output: 50
w2(70); // output: 30
w2(40); // output: "Insufficient funds"
w1(40); // output: 10
```

Observe that `w1` and `w2` are completely independent objects, each with its own local state variable `balance`. Withdrawals from one do not affect the other.

We can also create objects that handle deposits as well as withdrawals, and thus we can represent simple bank accounts. Here is a function that returns a ‘bank-account object’ with a specified initial balance:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
```

⁶In contrast with `make_withdraw` above, we do not have to use `let` to make `balance` a local variable, since formal parameters are already local. This will be clearer after the discussion of the environment model of evaluation in section 3.2. (See also exercise 3.10.)

```

        balance = balance + amount;
        return balance;
    }
    function dispatch(m) {
        if (m === "withdraw") {
            return withdraw;
        } else if (m === "deposit") {
            return deposit;
        } else {
            return "Unknown request - - MAKE-ACCOUNT";
        }
    }
    return dispatch;
}

```

Each call to `make_account` sets up an environment with a local state variable `balance`. Within this environment, `make_account` defines functions `deposit` and `withdraw` that access `balance` and an additional function `dispatch` that takes a ‘message’ as input and returns one of the two local functions. The `dispatch` function itself is returned as the value that represents the bank-account object. This is precisely the *message-passing* style of programming that we saw in section 2.4.3, although here we are using it in conjunction with the ability to modify local variables.

`make_account` can be used as follows:

```

const acc = make_account(100);

(acc("withdraw"))(50);

(acc("withdraw"))(60);

(acc("deposit"))(40);

(acc("withdraw"))(60);

```

Each call to `acc` returns the locally defined `deposit` or `withdraw` function, which is then applied to the specified `amount`. As was the case with `make_withdraw`, another call to `make_account`

```
const acc2 = make_account(100);
```

will produce a completely separate account object, which maintains its own local `balance`.

Exercise 3.1

An *accumulator* is a function that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a function `make_accumulator` that generates accumulators, each maintaining an

independent sum. The input to `make_accumulator` should specify the initial value of the sum; for example

```
// make_accumulator to be written by students
const a = make_accumulator(5);

a(10); // output: 15

a(10); // output: 25
```

[Solution](#)

Exercise 3.2

In software-testing applications, it is useful to be able to count the number of times a given function is called during the course of a computation. Write a function `make_monitored` that takes as input a function, `f`, that itself takes one input. The result returned by `make_monitored` is a third function, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the string "how many calls?", then `mf` returns the value of the counter. If the input is the string "reset count", then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `sqrt` function:

```
// make_monitored function to be written by students
const s = make_monitored(math_sqrt);

s(100);

s("how many calls?"); // returns 1
```

[Solution](#)

Exercise 3.3

Modify the `make_account` function so that it creates password-protected accounts. That is, `make_account` should take a symbol as an additional argument, as in

```
// make_account function to be written by students
const acc = make_account(100, "secret password");
```

The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

```
(acc("secret password", "withdraw"))(40); // result: 60
```

```
(acc("some other password", "deposit"))(40);
// result: incorrect password
```

[Solution](#)

Exercise 3.4

Modify the `make_account` function of exercise 3.3 by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the function `call_the_cops`.

[Solution](#)

3.1.2 The Benefits of Introducing Assignment

As we shall see, introducing assignment into our programming language leads us into a thicket of difficult conceptual issues. Nevertheless, viewing systems as collections of objects with local state is a powerful technique for maintaining a modular design. As a simple example, consider the design of a function `rand` that, whenever it is called, returns an integer chosen at random.

It is not at all clear what is meant by ‘chosen at random.’ What we presumably want is for successive calls to `rand` to produce a sequence of numbers that has statistical properties of uniform distribution. We will not discuss methods for generating suitable sequences here. Rather, let us assume that we have a function `rand_update` that has the property that if we start with a given number `x_1` and form

```
x_2 = rand_update(x_1);
x_3 = rand_update(x_2);
```

then the sequence of values `x_1, x_2, x_3, ...`, will have the desired statistical properties.⁷

We can implement `rand` as a function with a local state variable `x` that is initialized to some fixed value `random_init`. Each call to `rand` computes `rand_update` of the current value of `x`, returns this as the random number, and also stores this as the new value of `x`.

⁷One common way to implement `rand_update` is to use the rule that x is updated to $ax + b$ modulo m , where a , b , and m are appropriately chosen integers. Chapter 3 of Knuth 1981 includes an extensive discussion of techniques for generating sequences of random numbers and establishing their statistical properties. Notice that the `rand_update` function computes a mathematical function: Given the same input twice, it produces the same output. Therefore, the number sequence produced by `rand_update` certainly is not ‘random,’ if by ‘random’ we insist that each number in the sequence is unrelated to the preceding number. The relation between ‘real randomness’ and so-called *pseudo-random* sequences, which are produced by well-determined computations and yet have suitable statistical properties, is a complex question involving difficult issues in mathematics and philosophy. Kolmogorov, Solomonoff, and Chaitin have made great progress in clarifying these issues; a discussion can be found in Chaitin 1975.

```

function make_rand() {
    let x = random_init;
    function rand() {
        x = rand_update(x);
        return x;
    }
    return rand;
}
const rand = make_rand();

```

Of course, we could generate the same sequence of random numbers without using assignment by simply calling `rand_update` directly. However, this would mean that any part of our program that used random numbers would have to explicitly remember the current value of `x` to be passed as an argument to `rand_update`. To realize what an annoyance this would be, consider using random numbers to implement a technique called *Monte Carlo simulation*.

The Monte Carlo method consists of choosing sample experiments at random from a large set and then making deductions on the basis of the probabilities estimated from tabulating the results of those experiments. For example, we can approximate π using the fact that $6/\pi^2$ is the probability that two integers chosen at random will have no factors in common; that is, that their greatest common divisor will be 1.⁸ To obtain the approximation to π , we perform a large number of experiments. In each experiment we choose two integers at random and perform a test to see if their GCD is 1. The fraction of times that the test is passed gives us our estimate of $6/\pi^2$, and from this we obtain our approximation to π .

The heart of our program is a function `monte_carlo`, which takes as arguments the number of times to try an experiment, together with the experiment, represented as a no-argument function that will return either true or false each time it is run. `monte_carlo` runs the experiment for the designated number of trials and returns a number telling the fraction of the trials in which the experiment was found to be true.

```

function estimate_pi(trials) {
    return math_sqrt(6 / monte_carlo(trials, cesaro_test));
}

function cesaro_test() {
    return gcd(rand(), rand()) === 1;
}
function monte_carlo(trials, experiment) {
    function iter(trials_remaining, trials_passed) {
        if (trials_remaining === 0) {
            return trials_passed / trials;
        } else if (experiment()) {
            return iter(trials_remaining - 1,

```

⁸This theorem is due to E. Cesàro. See section 4.5.2 of Knuth 1981 for a discussion and a proof.

```

        trials_passed + 1);
    } else {
        return iter(trials_remaining - 1,
                    trials_passed);
    }
}
return iter(trials, 0);
}

```

Now let us try the same computation using `rand_update` directly rather than `rand`, the way we would be forced to proceed if we did not use assignment to model local state:

```

function estimate_pi(trials) {
    return math_sqrt(6 / random_gcd_test(trials, random_init));
}

function random_gcd_test(trials, initial_x) {
    function iter(trials_remaining, trials_passed, x) {
        const x1 = rand_update(x);
        const x2 = rand_update(x1);
        if (trials_remaining === 0) {
            return trials_passed / trials;
        } else if (gcd(x1, x2) === 1) {
            return iter(trials_remaining - 1,
                        trials_passed + 1, x2);
        } else {
            return iter(trials_remaining - 1,
                        trials_passed, x2);
        }
    }
    return iter(trials, 0, initial_x);
}

```

While the program is still simple, it betrays some painful breaches of modularity. In our first version of the program, using `rand`, we can express the Monte Carlo method directly as a general `monte_carlo` function that takes as an argument an arbitrary `experiment` function. In our second version of the program, with no local state for the random-number generator, `random_gcd_test` must explicitly manipulate the random numbers `x1` and `x2` and recycle `x2` through the iterative loop as the new input to `rand_update`. This explicit handling of the random numbers intertwines the structure of accumulating test results with the fact that our particular experiment uses two random numbers, whereas other Monte Carlo experiments might use one random number or three. Even the top-level function `estimate_pi` has to be concerned with supplying an initial random number. The fact that the random-number generator's insides are leaking out into other parts of the program makes it difficult for us to isolate the Monte Carlo idea so that it can be applied to other tasks. In the first version of the

program, assignment encapsulates the state of the random-number generator within the `rand` function, so that the details of random-number generation remain independent of the rest of the program.

The general phenomenon illustrated by the Monte Carlo example is this: From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write computer programs whose structure reflects this decomposition, we make computational objects (such as bank accounts and random-number generators) whose behavior changes with time. We model state with local state variables, and we model the changes of state with assignments to those variables.

It is tempting to conclude this discussion by saying that, by introducing assignment and the technique of hiding state in local variables, we are able to structure systems in a more modular fashion than if all state had to be manipulated explicitly, by passing additional parameters. Unfortunately, as we shall see, the story is not so simple.

Exercise 3.5

Monte Carlo integration is a method of estimating definite integrals by means of Monte Carlo simulation. Consider computing the area of a region of space described by a predicate $P(x, y)$ that is true for points (x, y) in the region and false for points not in the region. For example, the region contained within a circle of radius 3 centered at $(5, 7)$ is described by the predicate that tests whether $(x - 5)^2 + (y - 7)^2 \leq 3^2$. To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at $(2, 4)$ and $(8, 10)$ contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region. We can estimate the integral by picking, at random, points (x, y) that lie in the rectangle, and testing $P(x, y)$ for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

Implement Monte Carlo integration as a function `estimate_integral` that takes as arguments a predicate `P`, upper and lower bounds `x1`, `x2`, `y1`, and `y2` for the rectangle, and the number of trials to perform in order to produce the estimate. Your function should use the same `monte_carlo` function that was used above to estimate π . Use your `estimate_integral` to produce an estimate of π by measuring the area of a unit circle.

You will find it useful to have a function that returns a number chosen at random from a given range. The following `random_in_range` function implements this in terms of the `random` function used in section 1.2.6, which returns a nonnegative number less than its input.

```
function random_in_range(low, high) {
```

```

const range = high - low;
return low + random(range);
}

```

Exercise 3.6

It is useful to be able to reset a random-number generator to produce a sequence starting from a given value. Design a new `rand` function that is called with an argument that is either the symbol `generate` or the symbol `reset` and behaves as follows: `rand("generate")` produces a new random number; `(rand("reset"))(new-value)` resets the internal state variable to the designated *new-value*. Thus, by resetting the state, one can generate repeatable sequences. These are very handy to have when testing and debugging programs that use random numbers.

3.1.3 The Costs of Introducing Assignment

As we have seen, the assignment statement enables us to model objects that have local state. However, this advantage comes at a price. Our programming language can no longer be interpreted in terms of the substitution model of function application that we introduced in section 1.1.5. Moreover, no simple model with ‘nice’ mathematical properties can be an adequate framework for dealing with objects and assignment in programming languages.

So long as we do not use assignments, two evaluations of the same function with the same arguments will produce the same result, so that functions can be viewed as computing mathematical functions. Programming without any use of assignments, as we did throughout the first two chapters of this book, is accordingly known as *functional programming*.

To understand how assignment complicates matters, consider a simplified version of the `make_withdraw` function of section 3.1.1 that does not bother to check for an insufficient amount:

```

function make_simplified_withdraw(balance) {
    return amount => {
        balance = balance - amount;
        return balance;
    };
}

```

Compare this function with the following `make_decremter` function, which does not use assignment:

```

function make_decremter(balance) {
    return amount => balance - amount;
}

```

The function `make_decremter` returns a function that subtracts its input from a designated

amount balance, but there is no accumulated effect over successive calls, as with `make_simplified_withdraw`:

```
const d = make_decrementer(25);

d(20); // output: 5

d(10); // output: 15
```

We can use the substitution model to explain how `make_decrementer` works. For instance, let us analyze the evaluation of the expression

```
(make_decrementer(25))(20);
```

We first simplify the operator of the combination by substituting 25 for balance in the body of `make-decrementer`. This reduces the expression to

```
(amount => 25 - amount)(20);
```

Now we apply the operator by substituting 20 for amount in the body of the function definition expression:

```
25 - 20;
```

The final answer is 5.

Observe, however, what happens if we attempt a similar substitution analysis with `make_simplified_withdraw`:

```
(make_simplified_withdraw(25))(20);
```

We first simplify the operator by substituting 25 for balance in the return expression of `make_simplified_withdraw`. This reduces the expression to⁹

```
(amount => {
    balance = 25 - amount;
    return 25;
})(20);
```

Now we apply the function by substituting 20 for amount in the body of the function:

```
balance = 25 - 20;
return 25;
```

If we adhered to the substitution model, we would have to say that the meaning of the function application is to first set `balance` to 5 and then return 25 as the value of the expression. This gets the wrong answer. In order to get the correct answer, we would have to somehow distinguish the first occurrence of `balance` (before the effect of the assignment) from the second occurrence of `balance` (after the effect of the assignment), and the substitution model cannot do this.

⁹We don't substitute for the occurrence of `balance` in the assignment statement because the name in an assignment is not evaluated. If we did substitute for it, we would get `25 = 25 - amount;`, which makes no sense.

The trouble here is that substitution is based ultimately on the notion that the symbols in our language are essentially names for values. But as soon as we introduce assignment and the idea that the value of a variable can change, a variable can no longer be simply a name. Now a variable somehow refers to a place where a value can be stored, and the value stored at this place can change. In section 3.2 we will see how environments play this role of ‘place’ in our computational model.

Sameness and change

The issue surfacing here is more profound than the mere breakdown of a particular model of computation. As soon as we introduce change into our computational models, many notions that were previously straightforward become problematical. Consider the concept of two things being ‘the same’.

Suppose we call `make_decrementer` twice with the same argument to create two functions:

```
const d1 = make_decrementer(25);
const d2 = make_decrementer(25);
```

Are `d1` and `d2` the same? An acceptable answer is yes, because `d1` and `d2` have the same computational behavior—each is a function that subtracts its input from 25. In fact, `d1` could be substituted for `d2` in any computation without changing the result.

Contrast this with making two calls to `make_simplified_withdraw`:

```
const w1 = make_simplified_withdraw(25);
const w2 = make_simplified_withdraw(25);
```

Are `w1` and `w2` the same? Surely not, because calls to `w1` and `w2` have distinct effects, as shown by the following sequence of interactions:

```
w1(20);
w1(20);
w2(20);
```

Even though `w1` and `w2` are ‘equal’ in the sense that they are both created by evaluating the same expression, `make_simplified_withdraw(25)`, it is not true that `w1` could be substituted for `w2` in any expression without changing the result of evaluating the expression.

A language that supports the concept that ‘equals can be substituted for equals’ in an expression without changing the value of the expression is said to be *referentially transparent*. Referential transparency is violated when we include assignment in our computer language. This makes it tricky to determine when we can simplify expressions by substituting equivalent

expressions. Consequently, reasoning about programs that use assignment becomes drastically more difficult.

Once we forgo referential transparency, the notion of what it means for computational objects to be ‘the same’ becomes difficult to capture in a formal way. Indeed, the meaning of ‘same’ in the real world that our programs model is hardly clear in itself. In general, we can determine that two apparently identical objects are indeed ‘the same one’ only by modifying one object and then observing whether the other object has changed in the same way. But how can we tell if an object has ‘changed’ other than by observing the ‘same’ object twice and seeing whether some property of the object differs from one observation to the next? Thus, we cannot determine ‘change’ without some *a priori* notion of ‘sameness,’ and we cannot determine sameness without observing the effects of change.

As an example of how this issue arises in programming, consider the situation where Peter and Paul have a bank account with \$100 in it. There is a substantial difference between modeling this as

```
const peter_acc = make_account(100);
const paul_acc = make_account(100);
```

and modeling it as

```
const peter_acc = make_account(100);
const paul_acc = peter_acc;
```

In the first situation, the two bank accounts are distinct. Transactions made by Peter will not affect Paul’s account, and vice versa. In the second situation, however, we have defined paul_acc to be *the same thing* as peter_acc. In effect, Peter and Paul now have a joint bank account, and if Peter makes a withdrawal from peter_acc Paul will observe less money in paul_acc. These two similar but distinct situations can cause confusion in building computational models. With the shared account, in particular, it can be especially confusing that there is one object (the bank account) that has two different names (peter_acc and paul_acc); if we are searching for all the places in our program where paul_acc can be changed, we must remember to look also at things that change peter_acc.¹⁰

With reference to the above remarks on ‘sameness’ and ‘change,’ observe that if Peter and Paul could only examine their bank balances, and could not perform operations that changed the balance, then the issue of whether the two accounts are distinct would be moot. In general, so long as we never modify data objects, we can regard a compound data object to be precisely

¹⁰The phenomenon of a single computational object being accessed by more than one name is known as *aliasing*. The joint bank account situation illustrates a very simple example of an alias. In section 3.3 we will see much more complex examples, such as ‘distinct’ compound data structures that share parts. Bugs can occur in our programs if we forget that a change to an object may also, as a ‘side effect,’ change a ‘different’ object because the two ‘different’ objects are actually a single object appearing under different aliases. These so-called *side-effect bugs* are so difficult to locate and to analyze that some people have proposed that programming languages be designed in such a way as to not allow side effects or aliasing (Lampson et al. 1981; Morris, Schmidt, and Wadler 1980).

the totality of its pieces. For example, a rational number is determined by giving its numerator and its denominator. But this view is no longer valid in the presence of change, where a compound data object has an ‘identity’ that is something different from the pieces of which it is composed. A bank account is still ‘the same’ bank account even if we change the balance by making a withdrawal; conversely, we could have two different bank accounts with the same state information. This complication is a consequence, not of our programming language, but of our perception of a bank account as an object. We do not, for example, ordinarily regard a rational number as a changeable object with identity, such that we could change the numerator and still have ‘the same’ rational number.

Pitfalls of imperative programming

In contrast to functional programming, programming that makes extensive use of assignment is known as *imperative programming*. In addition to raising complications about computational models, programs written in imperative style are susceptible to bugs that cannot occur in functional programs. For example, recall the iterative factorial program from section 1.2.1:

```
function factorial(n) {
    function iter(product,counter) {
        if (counter > n) {
            return product;
        } else {
            return iter(counter*product,
                       counter+1);
        }
    }
    return iter(1,1);
}
```

Instead of passing arguments in the internal iterative loop, we could adopt a more imperative style by using explicit assignment to update the values of the variables *product* and *counter*:

```
function factorial(n) {
    let product = 1;
    let counter = 1;
    function iter() {
        if (counter > n) {
            return product;
        } else {
            product = counter * product;
            counter = counter + 1;
            return iter();
        }
    }
    return iter();
```

```
}
```

This does not change the results produced by the program, but it does introduce a subtle trap. How do we decide the order of the assignments? As it happens, the program is correct as written. But writing the assignments in the opposite order

```
counter = counter + 1;
product = counter * product;
```

would have produced a different, incorrect result. In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs.¹¹

The complexity of imperative programs becomes even worse if we consider applications in which several processes execute concurrently. We will return to this in section 3.4. First, however, we will address the issue of providing a computational model for expressions that involve assignment, and explore the uses of objects with local state in designing simulations.

Exercise 3.7

Consider the bank account objects created by `make_account`, with the password modification described in exercise 3.3. Suppose that our banking system requires the ability to make joint accounts. Define a function `make_joint` that accomplishes this. The function `make_joint` should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the `make_joint` operation to proceed. The third argument is a new password. The function `make_joint` is to create an additional access to the original account using the new password. For example, if (`peter_acc` is a bank account with password ("open sesame", then

```
// make_joint function to be written by students
const paul_acc =
    make_joint(peter_acc, "open sesame", "rosebud");
```

will allow one to make transactions on (`peter_acc` using the name `paul_acc` and the password "rosebud". You may wish to modify your solution to exercise 3.3 to accommodate this new feature.

Exercise 3.8

¹¹In view of this, it is ironic that introductory programming is most often taught in a highly imperative style. This may be a vestige of a belief, common throughout the 1960s and 1970s, that programs that call functions must inherently be less efficient than programs that perform assignments. (Steele (1977) debunks this argument.) Alternatively it may reflect a view that step-by-step assignment is easier for beginners to visualize than function call. Whatever the reason, it often saddles beginning programmers with 'should I set this variable before or after that one' concerns that can complicate programming and obscure the important ideas.

When we defined the evaluation model in section 1.1.3, we said that the first step in evaluating an expression is to evaluate its subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a function are evaluated can make a difference to the result. Define a simple function f such that evaluating $f(0) + f(1)$ will return 0 if the arguments to $+$ are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

3.2 The Environment Model of Evaluation

When we introduced compound functions in chapter 1, we used the substitution model of evaluation (section 1.1.5) to define what is meant by applying a function to arguments:

- To apply a compound function to arguments, evaluate the body of the function with each formal parameter replaced by the corresponding argument.

Once we admit assignment into our programming language, such a definition is no longer adequate. In particular, section 3.1.3 argued that, in the presence of assignment, a variable can no longer be considered to be merely a name for a value. Rather, a variable must somehow designate a ‘place’ in which values can be stored. In our new model of evaluation, these places will be maintained in structures called *environments*.

An environment is a sequence of *frames*. Each frame is a table (possibly empty) of *bindings*, which associate variable names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its *enclosing environment*, unless, for the purposes of discussion, the frame is considered to be *global*. The *value of a variable* with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be *unbound* in the environment.

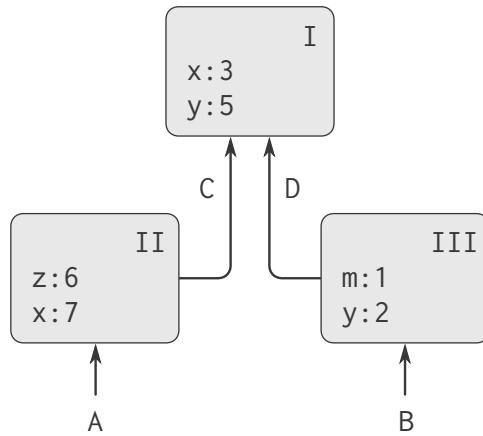


Figure 3.1: A simple environment structure.

Figure 3.1 shows a simple environment structure consisting of three frames, labeled I, II, and III. In the diagram, A, B, C, and D are pointers to environments. C and D point to the same environment. The variables z and x are bound in frame II, while y and x are bound in frame I. The value of x in environment D is 3. The value of x with respect to environment B is also 3. This is determined as follows: We examine the first frame in the sequence (frame III) and do not find a binding for x , so we proceed to the enclosing environment D and find the binding in frame I. On the other hand, the value of x in environment A is 7, because the first frame in the sequence (frame II) contains a binding of x to 7. With respect to environment A, the binding of x to 7 in frame II is said to *shadow* the binding of x to 3 in frame I.

The environment is crucial to the evaluation process, because it determines the context in which an expression should be evaluated. Indeed, one could say that expressions in a programming language do not, in themselves, have any meaning. Rather, an expression acquires a meaning only with respect to some environment in which it is evaluated. Even the interpretation of an expression as straightforward as `display(1)` depends on an understanding that one is operating in a context in which `display` refers to the primitive function that displays a value. Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment. To describe interactions with the interpreter, we will suppose that there is a global environment, consisting of a single frame (with no enclosing environment) that includes values for the symbols associated with the primitive functions. For example, the idea that `display` is the primitive function for displaying a value is captured by saying that the variable `display` is bound in the global environment to the respective primitive function.

3.2.1 The Rules for Evaluation

The overall specification of how the interpreter evaluates an application combination remains the same as when we first introduced it in section 1.1.5:

- To evaluate an application combination of the form
`function-expression (argument-expressions)`

do the following:

- Evaluate the function expression of the application combination, resulting in the function to be applied.
- Evaluate the argument expressions of the combination.
- Apply the function to the arguments.

The environment model of evaluation replaces the substitution model in specifying what it means to apply a compound function to arguments.

In the environment model of evaluation, a function is always a pair consisting of some code and a pointer to an environment. Functions are created in one way only: by evaluating a function definition expression. This produces a function whose code is obtained from the text of the function definition expression and whose environment is the environment in which the function definition expression was evaluated to produce the function. For example, consider the function declaration

```
function square(x) {
    return x * x;
}
```

evaluated in the global environment. The function declaration syntax is just syntactic sugar for an underlying implicit function definition expression. It would have been equivalent to have used

```
const square = x => x * x;
```

which evaluates `x => x * x` and binds `square` to the resulting value, all in the global environment.

Figure 3.2 shows the result of evaluating this function declaration statement. The function object is a pair whose code specifies that the function has one formal parameter, namely `x`, and a function body `return x * x;`. The environment part of the function is a pointer to the global environment, since that is the environment in which the function definition expression was evaluated to produce the function. A new binding, which associates the function object with the symbol `square`, has been added to the global frame. In general, `const` and `let` create declarations by adding bindings to frames.

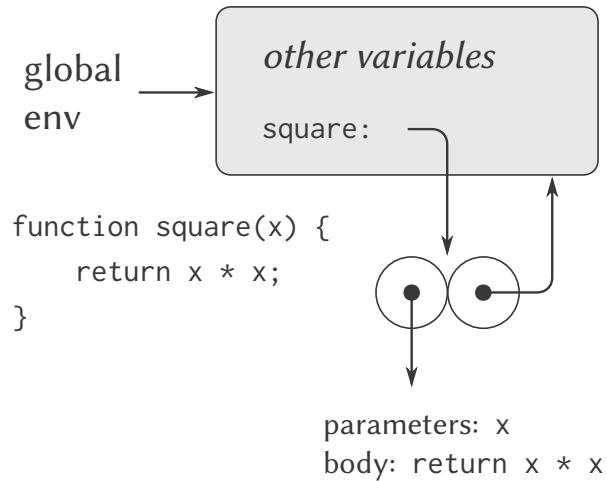


Figure 3.2: Environment structure produced by evaluating `function square(x) return x * x;` in the global environment.

Now that we have seen how functions are created, we can describe how functions are applied. The environment model specifies: To apply a function to arguments, create a new environment containing a frame that binds the parameters to the values of the arguments. The enclosing environment of this frame is the environment specified by the function. Now, within this new environment, evaluate the function body.

To show how this rule is followed, Figure 3.3 illustrates the environment structure created by evaluating the expression `square(5)`; in the global environment, where `square` is the function generated in Figure 3.2. Applying the function results in the creation of a new environment, labeled E1 in the figure, that begins with a frame in which `x`, the formal parameter for the function, is bound to the argument 5. The pointer leading upward from this frame shows that the frame's enclosing environment is the global environment. The global environment is chosen here, because this is the environment that is indicated as part of the `square` function object. Within E1, we evaluate the body of the function, `return x * x;`. Since the value of `x` in E1 is 5, the result is $5 * 5$, or 25.

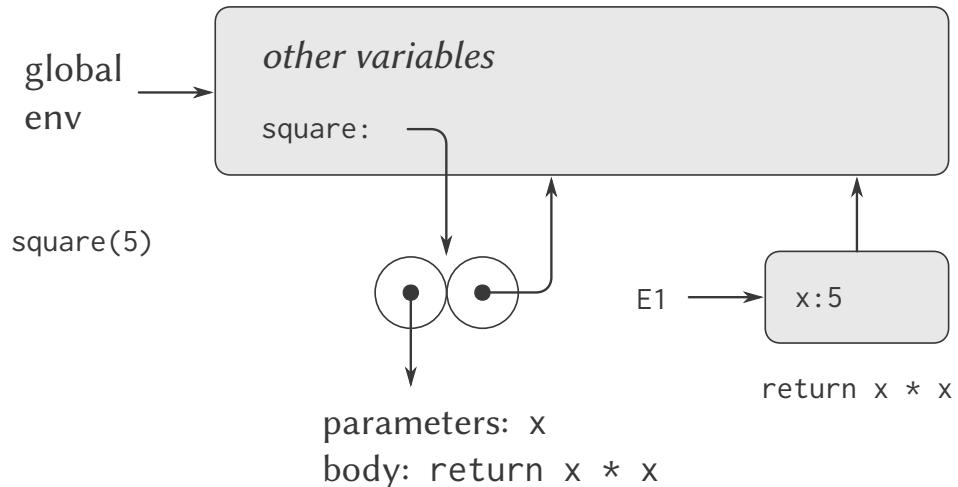


Figure 3.3: Environment created by evaluating `square(5)`; in the global environment.

The environment model of function application can be summarized by two rules:

- A function object is applied to a set of arguments by constructing a frame, in which we create variable bindings of the parameters of the function to the arguments of the call, and then evaluating the body of the function in the context of the new environment constructed. The new frame has as its enclosing environment the environment part of the function object being applied.
- A function is created by evaluating a function definition expression relative to a given environment. The resulting function object is a pair consisting of the text of the function definition expression and a pointer to the environment in which the function was created.

We also specify that defining a symbol using **const/let** creates a constant/variable binding in the current environment frame and assigns to the symbol the indicated value. Finally, we specify the behavior of assignment, the operation that forced us to introduce the environment model in the first place. Evaluating the statement `name = value;` in some environment locates the binding of the name in the environment. For this, one finds the first frame in the environment that contains a binding for the name. If the name is unbound in the environment, then the assignment signals a ‘variable undefined’ error. Otherwise, if the binding in the frame is a constant binding, the assignment signals an ‘assignment to constant’ error, because JavaScript forbids assignment to constants. At last, if the binding in the frame is a variable binding, that binding is changed to reflect the new value of the variable.

These evaluation rules, though considerably more complex than the substitution model, are still reasonably straightforward. Moreover, the evaluation model, though abstract, provides a correct description of how the interpreter evaluates expressions. In chapter 4 we shall see how this model can serve as a blueprint for implementing a working interpreter. The following sections elaborate the details of the model by analyzing some illustrative programs.

3.2.2 Applying Simple Functions

When we introduced the substitution model in section 1.1.5 we showed how the combination $f(5)$ evaluates to 136, given the following function definitions:

```
function square(x) {
    return x * x;
}

function sum_of_squares(x, y) {
    return square(x) + square(y);
}

function f(a) {
    return sum_of_squares(a + 1, a * 2);
}
```

We can analyze the same example using the environment model. Figure 3.4 shows the three function objects created by evaluating the definitions of f , square , and sum_of_squares in the global environment. Each function object consists of some code, together with a pointer to the global environment.

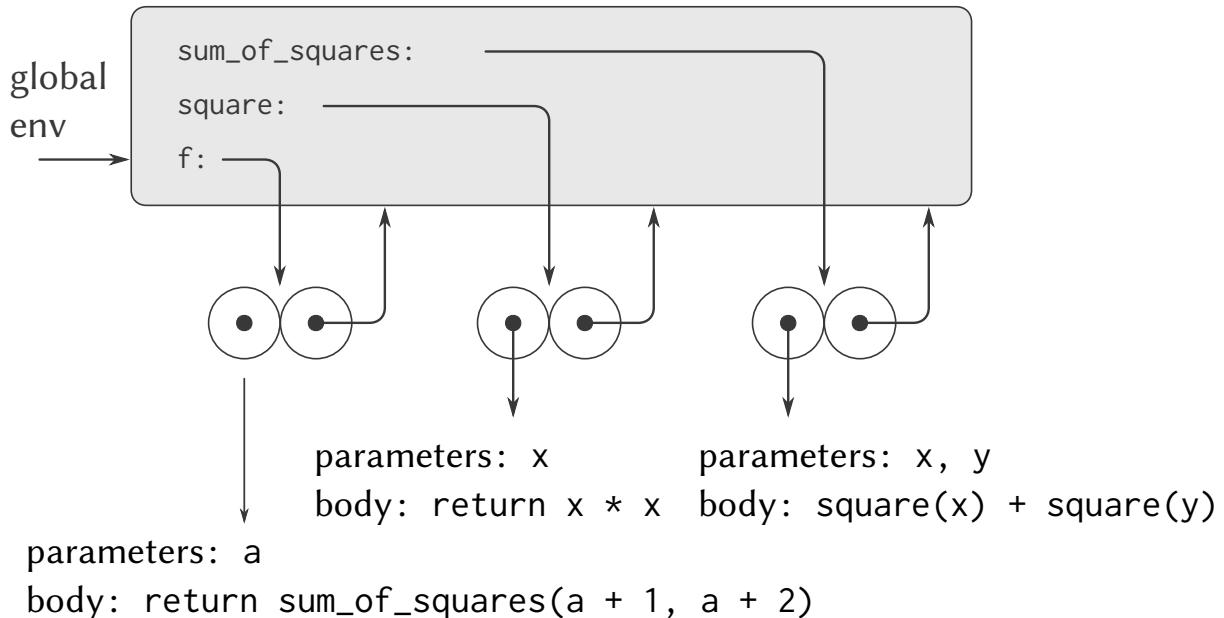


Figure 3.4: Function objects in the global frame.

In Figure 3.5 we see the environment structure created by evaluating the expression $f(5)$. The call to f creates a new environment $E1$ beginning with a frame in which a , the formal parameter of f , is bound to the argument 5. In $E1$, we evaluate the body of f :

```
return sum_of_squares(a + 1, a * 2);
```

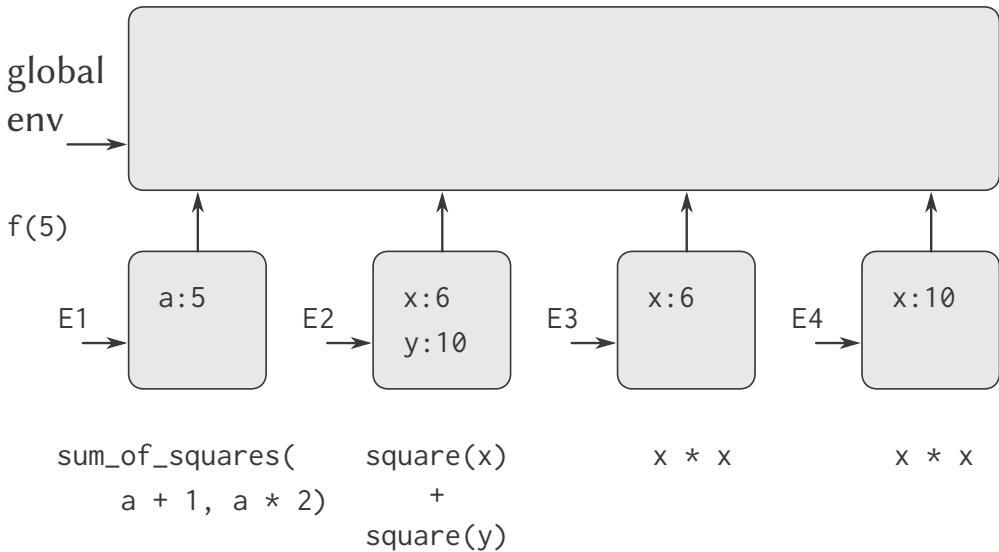


Figure 3.5: Environments created by evaluating $f(5)$ using the functions in Figure 3.4.

To evaluate this combination, we first evaluate the subexpressions. The first subexpression, `sum_of_squares`, has a value that is a function object. (Notice how this value is found: We first look in the first frame of E1, which contains no binding for `sum_of_squares`. Then we proceed to the enclosing environment, i.e. the global environment, and find the binding shown in Figure 3.4.) The other two subexpressions are evaluated by applying the primitive operations `+` and `*` to evaluate the two combinations $a + 1$ and $a * 2$ to obtain 6 and 10, respectively.

Now we apply the function object `sum_of_squares` to the arguments 6 and 10. This results in a new environment E2 in which the formal parameters x and y are bound to the arguments. Within E2 we evaluate the combination `square(x) + square(y)`. This leads us to evaluate `square(x)`, where `square` is found in the global frame and x is 6. Once again, we set up a new environment, E3, in which x is bound to 6, and within this we evaluate the body of `square`, which is $x * x$. Also as part of applying `sum_of_squares`, we must evaluate the subexpression `square(y)`, where y is 10. This second call to `square` creates another environment, E4, in which x , the formal parameter of `square`, is bound to 10. And within E4 we must evaluate $x * x$.

The important point to observe is that each call to `square` creates a new environment containing a binding for x . We can see here how the different frames serve to keep separate the different local variables all named x . Notice that each frame created by `square` points to the global environment, since this is the environment indicated by the `square` function object.

After the subexpressions are evaluated, the results are returned. The values generated by the two calls to `square` are added by `sum_of_squares`, and this result is returned by `f`. Since our focus here is on the environment structures, we will not dwell on how these returned values are passed from call to call; however, this is also an important aspect of the evaluation process, and we will return to it in detail in chapter 5.

Exercise 3.9

In section 1.2.1 we used the substitution model to analyze two functions for computing factorials, a recursive version

```
function factorial(n) {
    return n === 1
        ? 1
        : n * factorial(n - 1);
}
```

and an iterative version

```
function factorial(n) {
    return fact_iter(1, 1, n);
}
function fact_iter(product, counter, max_count) {
    return counter > max_count
        ? product
        : fact_iter(counter * product,
                    counter + 1,
                    max_count);
}
```

Show the environment structures created by evaluating `factorial(6)` using each version of the factorial function.¹²

3.2.3 Frames as the Repository of Local State

We can turn to the environment model to see how functions and assignment can be used to represent objects with local state. As an example, consider the ‘withdrawal processor’ from section 3.1.1 created by calling the function

```
function make_withdraw_with_balance(balance) {
    return amount => {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "insufficient funds";
        }
    };
}
```

Let us describe the evaluation of

¹²The environment model will not clarify our claim in section 1.2.1 that the interpreter can execute a function such as `fact_iter` in a constant amount of space using tail recursion. We will discuss tail recursion when we

```
const w1 = make_withdraw_with_balance(100);
```

followed by

```
w1(50);
```

Figure 3.6 shows the result of declaring the `make_withdraw_with_balance` function in the global environment. This produces a function object that contains a pointer to the global environment. So far, this is no different from the examples we have already seen, except that the body of the function is itself a function definition expression.

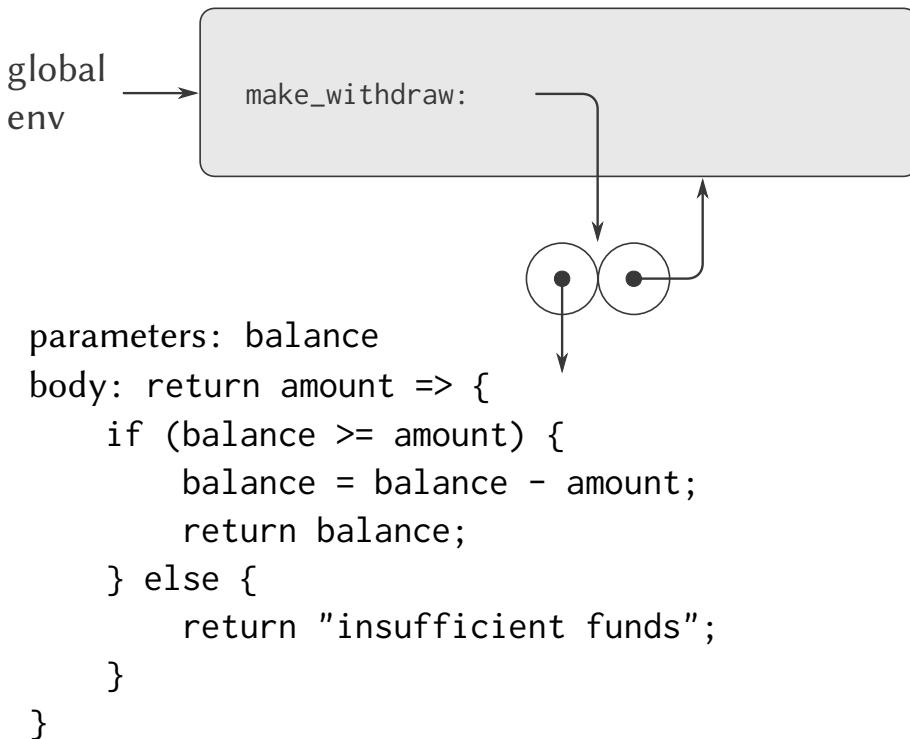


Figure 3.6: Result of defining `make_withdraw_with_balance` in the global environment.

The interesting part of the computation happens when we apply the function

```
make_withdraw_with_balance
```

to an argument:

```
const w1 = make_withdraw_with_balance(100);
```

We begin, as usual, by setting up an environment E1 in which the formal parameter `balance` is bound to the argument 100. Within this environment, we evaluate the body of `make_withdraw_with_balance`, namely the function definition expression. This constructs a new function object, whose code is as specified by the function definition and whose environment is E1, the environment in which the function definition was evaluated to produce the function. The resulting function object is the value returned by the call to `make_withdraw_with_balance`. This is bound to `w1` in the global environment, since the constant declaration itself is being evaluated in the global

environment. Figure 3.7 shows the resulting environment structure.

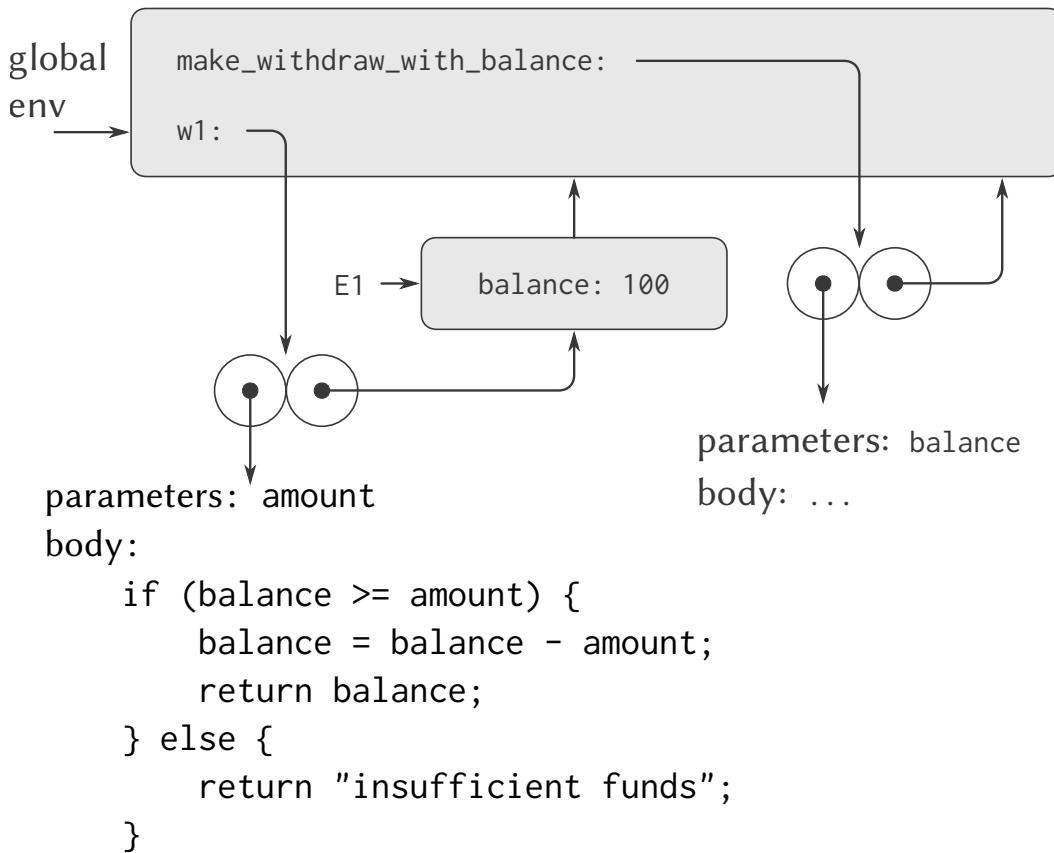


Figure 3.7: Result of evaluating `const w1 = make_withdraw_with_balance(100);.`

Now we can analyze what happens when `w1` is applied to an argument:

`w1(50);`

We begin by constructing a frame in which `amount`, the formal parameter of `w1`, is bound to the argument 50. The crucial point to observe is that this frame has as its enclosing environment not the global environment, but rather the environment `E1`, because this is the environment that is specified by the `w1` function object. Within this new environment, we evaluate the body of the function:

```

if (balance >= amount) {
  balance = balance - amount;
  return balance;
} else {
  return "insufficient funds";
}

```

The resulting environment structure is shown in Figure 3.8. The expression being evaluated references both `amount` and `balance`. The variable `amount` will be found in the first frame in the environment, while `balance` will be found by following the enclosing-environment pointer to `E1`.

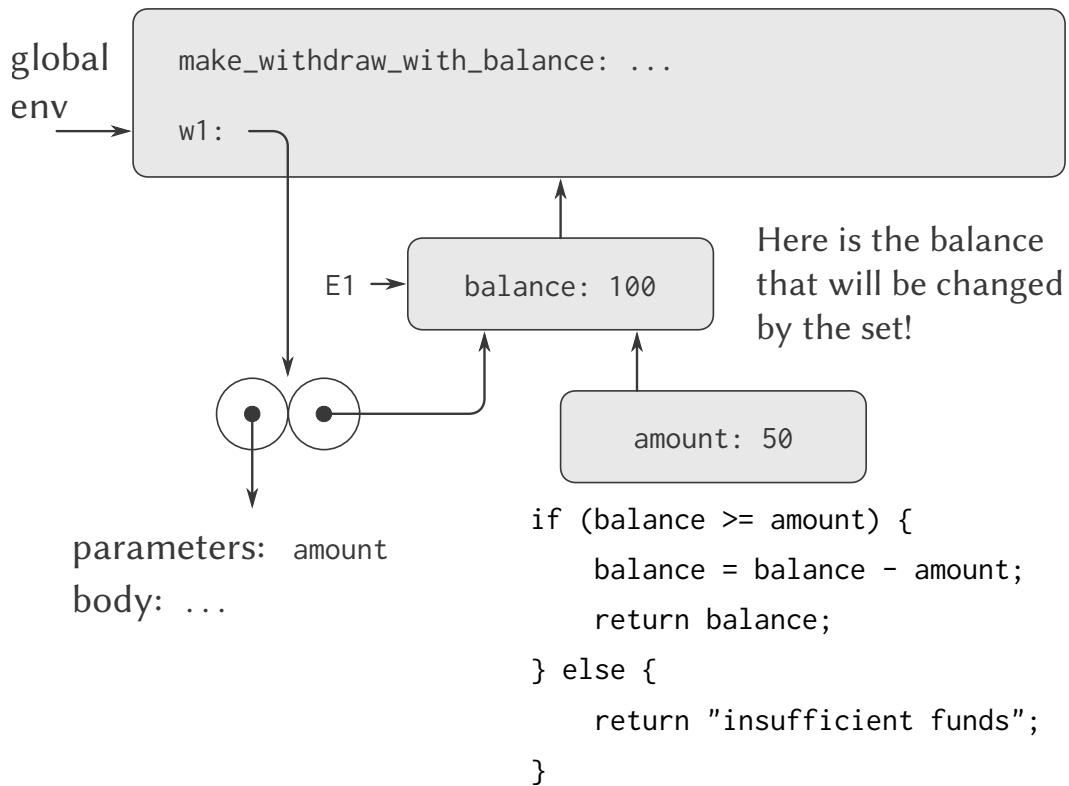
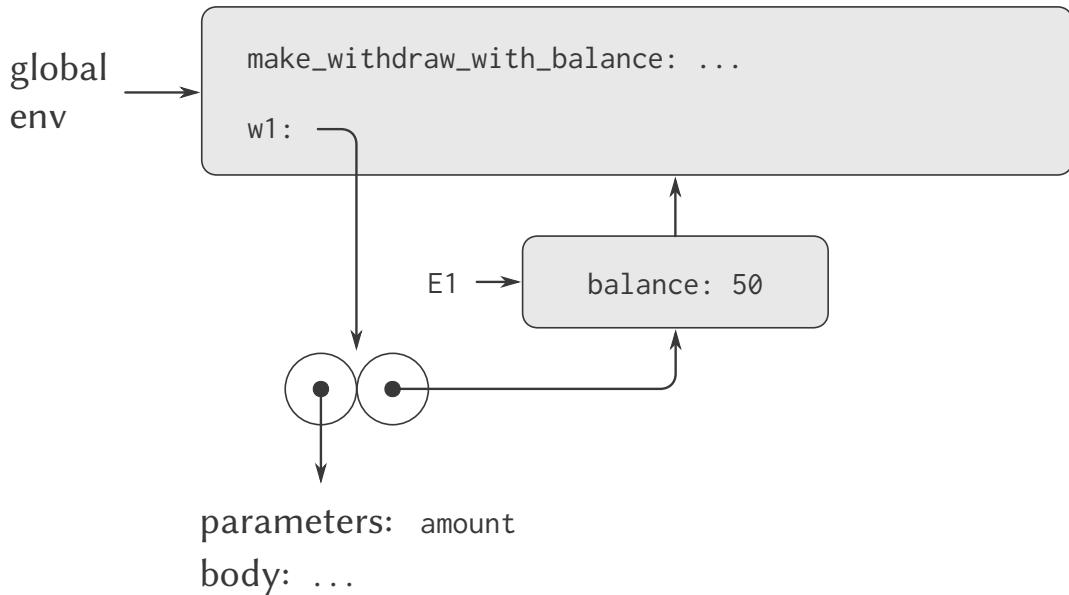


Figure 3.8: Environments created by applying the function object `w1`.

When the assignment is executed, the binding of `balance` in `E1` is changed. At the completion of the call to `w1`, `balance` is 50, and the frame that contains `balance` is still pointed to by the function object `w1`. The frame that binds `amount` (in which we executed the code that changed `balance`) is no longer relevant, since the function call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment. The next time `w1` is called, this will build a new frame that binds `amount` and whose enclosing environment is `E1`. We see that `E1` serves as the ‘place’ that holds the local state variable for the function object `w1`. Figure 3.9 shows the situation after the call to `W1`.

Figure 3.9: Environments after the call to `w1`.

Observe what happens when we create a second ‘withdraw’ object by making another call to `make_withdraw`:

```
const w2 = make_withdraw(100);
```

This produces the environment structure of Figure 3.10, which shows that `w2` is a function object, that is, a pair with some code and an environment. The environment `E2` for `w2` was created by the call to `make_withdraw`. It contains a frame with its own local binding for `balance`. On the other hand, `w1` and `w2` have the same code: the code specified by the function definition expression in the body of `make_withdraw`.¹³ We see here why `w1` and `w2` behave as independent objects. Calls to `w1` reference the state variable `balance` stored in `E1`, whereas calls to `w2` reference the `balance` stored in `E2`. Thus, changes to the local state of one object do not affect the other object.

¹³Whether `w1` and `w2` share the same physical code stored in the computer, or whether they each keep a copy of the code, is a detail of the implementation. For the interpreter we implement in chapter 4, the code is in fact shared.

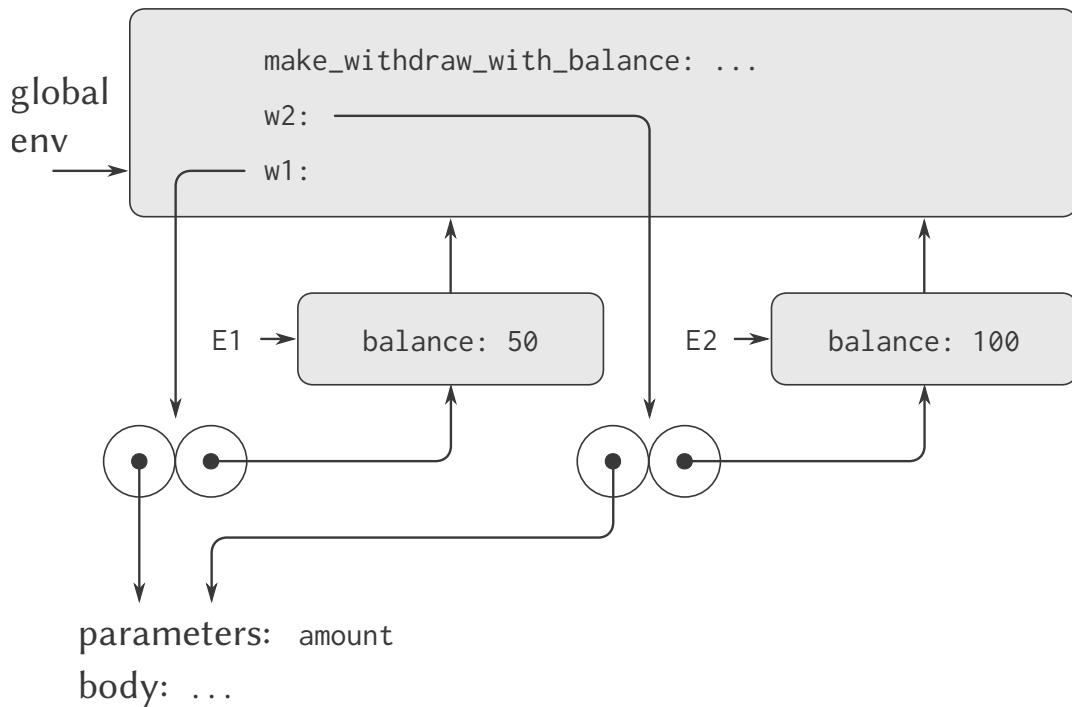


Figure 3.10: Using `const w2 = make_withdraw_with_balance(100)` to create a second object.

Exercise 3.10

In the `make_withdraw` function, the local variable `balance` is created as a parameter of `make_withdraw`. We could also create the local state variable explicitly, using `let`, as follows:

```
function make_withdraw(initial_amount) {
  let balance = initial_amount;
  function withdraw(amount) {
    if (balance >= amount) {
      balance = balance - amount;
      return balance;
    } else {
      return "insufficient funds";
    }
  }
  return withdraw;
}
```

Use the environment model to analyze this alternate version of `make_withdraw`, drawing figures like the ones above to illustrate the interactions

```
const w1 = make_withdraw(100);
w1(50);
const w2 = make_withdraw(100);
```

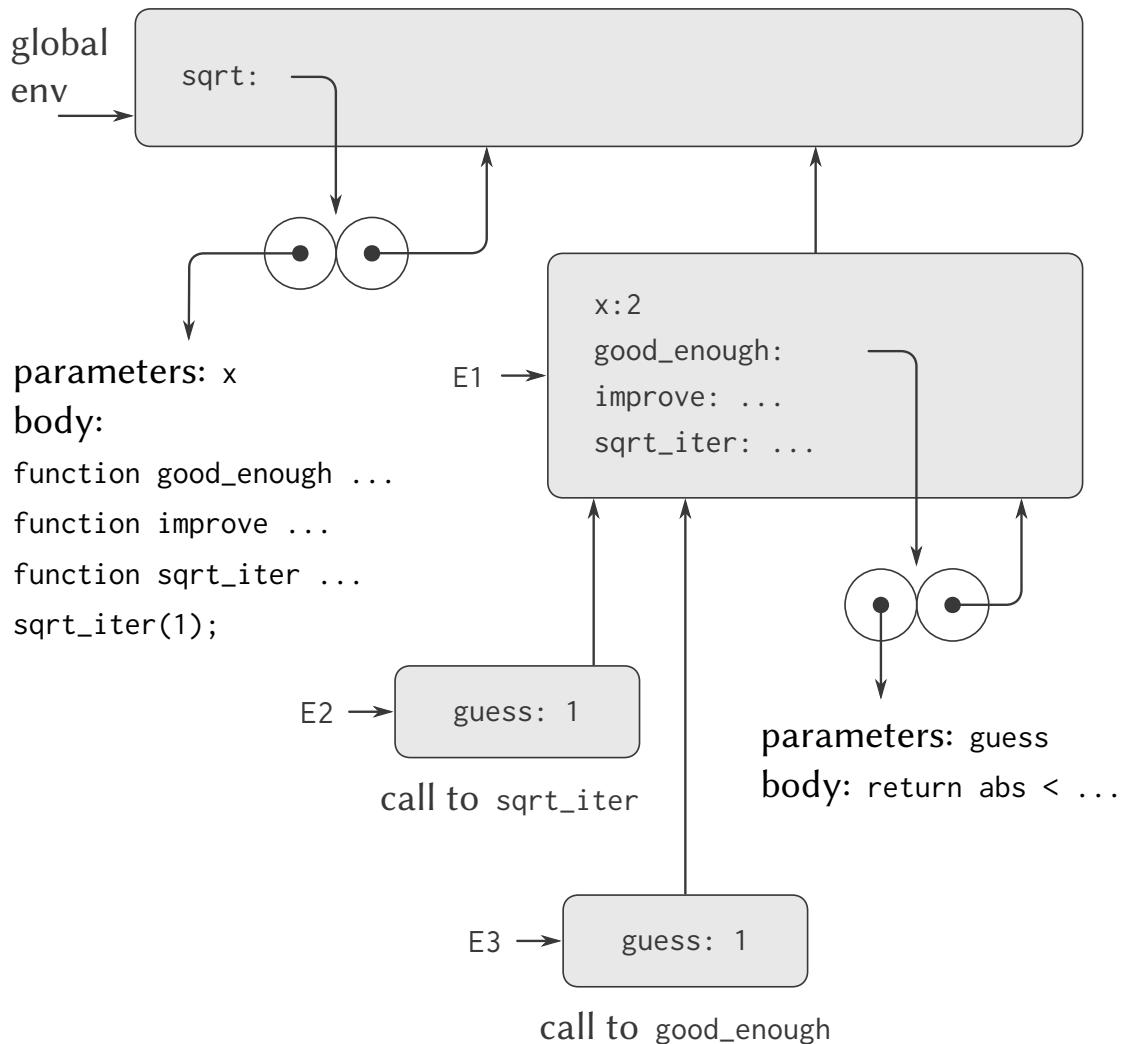
Show that the two versions of `make_withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?

3.2.4 Internal Definitions

Section 1.1.8 introduced the idea that functions can have internal definitions, thus leading to a block structure as in the following function to compute square roots:

```
function sqrt(x) {
    function good_enough(guess, x) {
        return abs(square(guess) - x) < 0.001;
    }
    function improve(guess) {
        return average(guess, x/guess);
    }
    function sqrt_iter(guess){
        if (good_enough(guess,x)) {
            return guess;
        } else {
            return sqrt_iter(improve(guess));
        }
    }
    return sqrt_iter(1.0);
}
```

Now we can use the environment model to see why these internal definitions behave as desired. Figure 3.11 shows the point in the evaluation of the expression `sqrt(2)` where the internal function `good_enough` has been called for the first time with `guess` equal to 1.

Figure 3.11: `sqrt` function with internal definitions.

Observe the structure of the environment. `sqrt` is a symbol in the global environment that is bound to a function object whose associated environment is the global environment. When `sqrt` was called, a new environment `E1` was formed, subordinate to the global environment, in which the parameter `x` is bound to 2. The body of `sqrt` was then evaluated in `E1`. Since the first expression in the body of `sqrt` is

```

function good_enough(guess) {
  return abs(square(guess) - x) < 0.001;
}
  
```

evaluating this expression defined the function `good_enough` in the environment `E1`. To be more precise, the symbol `good_enough` was added to the first frame of `E1`, bound to a function object whose associated environment is `E1`. Similarly, `improve` and `sqrt_iter` were defined as functions in `E1`. For conciseness, Figure 3.11 shows only the function object for `good_enough`.

After the local functions were defined, the expression `sqrt_iter(1.0)` was evaluated, still in environment `E1`. So the function object bound to `sqrt_iter` in `E1` was called with 1 as

an argument. This created an environment E2 in which `guess`, the parameter of `sqrt_iter`, is bound to 1. The function `sqrt_iter` in turn called `good_enough` with the value of `guess` (from E2) as the argument for `good_enough`. This set up another environment, E3, in which `guess` (the parameter of `good_enough`) is bound to 1. Although `sqrt_iter` and `good_enough` both have a parameter named `guess`, these are two distinct local variables located in different frames. Also, E2 and E3 both have E1 as their enclosing environment, because the `sqrt_iter` and `good_enough` functions both have E1 as their environment part. One consequence of this is that the symbol `x` that appears in the body of `good_enough` will reference the binding of `x` that appears in E1, namely the value of `x` with which the original `sqrt` function was called.

The environment model thus explains the two key properties that make local function definitions a useful technique for modularizing programs:

- The names of the local functions do not interfere with names external to the enclosing function, because the local function names will be bound in the frame that the function creates when it is run, rather than being bound in the global environment.
- The local functions can access the arguments of the enclosing function, simply by using parameter names as free variables. This is because the body of the local function is evaluated in an environment that is subordinate to the evaluation environment for the enclosing function.

Exercise 3.11

In section 3.2.3 we saw how the environment model described the behavior of functions with local state. Now we have seen how internal definitions work. A typical message-passing function contains both of these aspects. Consider the bank account function of section 3.1.1:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
    }
    function dispatch(m) {
        return m === "withdraw"
            ? withdraw
```

```

        : m === "deposit"
        ? deposit
        : "Unknown request: make_account";
    }
    return dispatch;
}

```

Show the environment structure generated by the sequence of interactions

```

const acc = make_account(50);
(acc("deposit"))(40);
(acc("withdraw"))(60);

```

Where is the local state for acc kept? Suppose we define another account

```
const acc2 = make_account(100);
```

How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between acc and acc2?

3.3 Modeling with Mutable Data

Chapter 2 dealt with compound data as a means for constructing computational objects that have several parts, in order to model real-world objects that have several aspects. In that chapter we introduced the discipline of data abstraction, according to which data structures are specified in terms of constructors, which create data objects, and selectors, which access the parts of compound data objects. But we now know that there is another aspect of data that chapter 2 did not address. The desire to model systems composed of objects that have changing state leads us to the need to modify compound data objects, as well as to construct and select from them. In order to model compound objects with changing state, we will design data abstractions to include, in addition to selectors and constructors, operations called *mutators*, which modify data objects. For instance, modeling a banking system requires us to change account balances. Thus, a data structure for representing bank accounts might admit an operation

```
set_balance(account, new-value)
```

that changes the balance of the designated account to the designated new value. Data objects for which mutators are defined are known as *mutable data objects*.

Chapter 2 introduced pairs as a general-purpose ‘glue’ for synthesizing compound data. We begin this section by defining basic mutators for pairs, so that pairs can serve as building blocks for constructing mutable data objects. These mutators greatly enhance the representational power of pairs, enabling us to build data structures other than the sequences and trees that we worked with in section 2.2. We also present some examples of simulations in which complex systems are modeled as collections of objects with local state.

3.3.1 Mutable List Structure

The basic operations on pairs—`pair`, `head`, and `tail`—can be used to construct list structure and to select parts from list structure, but they are incapable of modifying list structure. The same is true of the list operations we have used so far, such as `append` and `list`, since these can be defined in terms of `pair`, `head`, and `tail`. To modify list structures we need new operations.

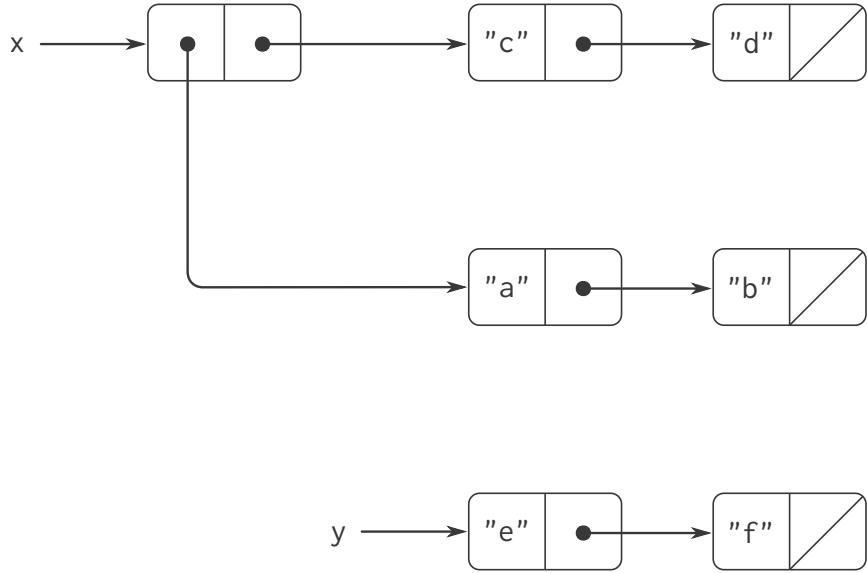


Figure 3.12: Lists `list(list("a", "b"), "c", "d")` and `y: list("e", "f")`.

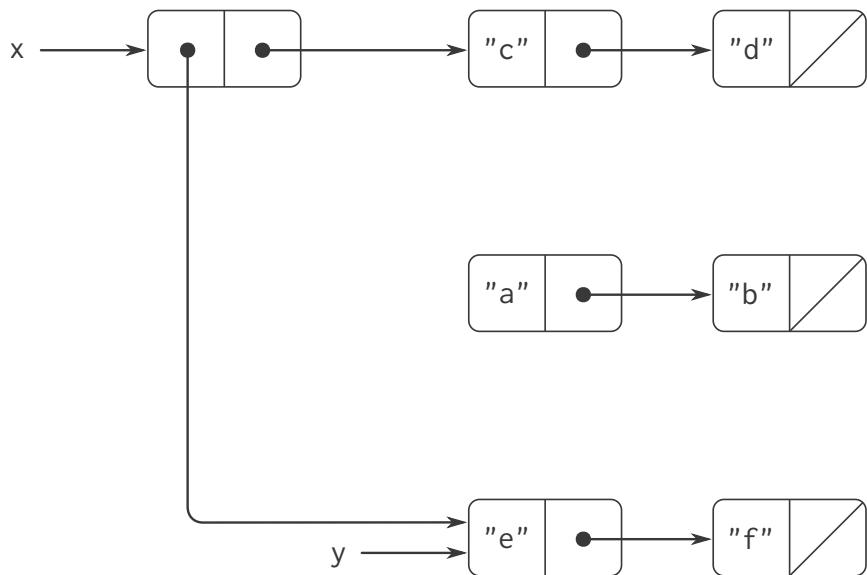
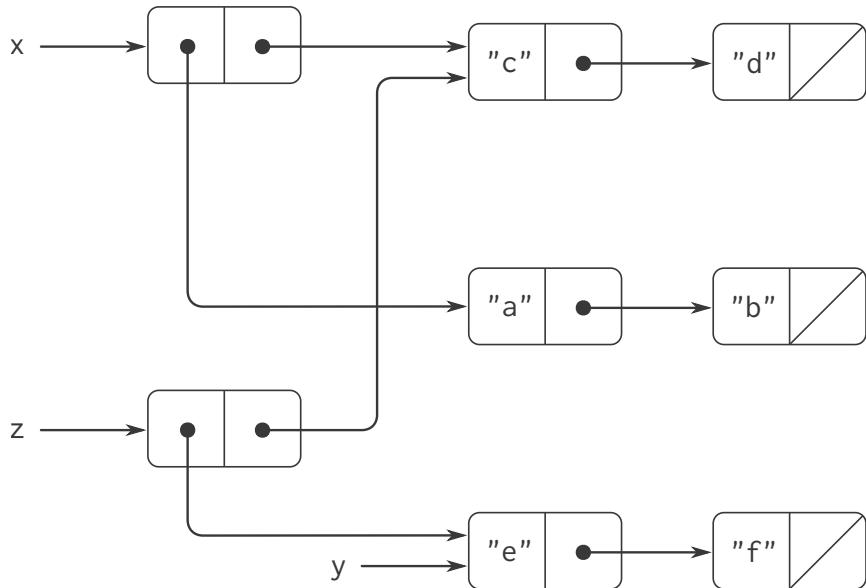
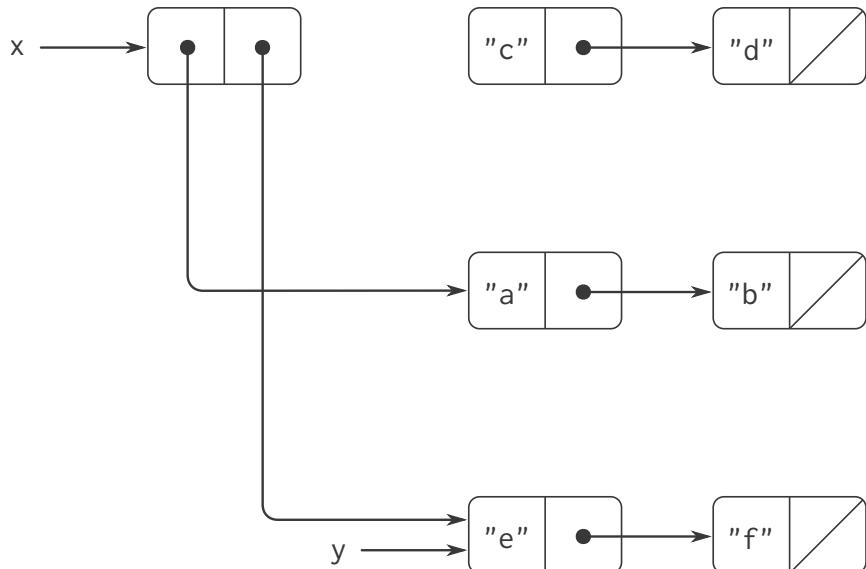


Figure 3.13: Effect of `set_head(x, y)` on the lists in figure 3.12.

Figure 3.14: Effect of `const z = pair(y, tail(x));` on the lists in figure 3.12.Figure 3.15: Effect of `set_tail(x, y)` on the lists in figure 3.12.

The primitive mutators for pairs are `set_head` and `set_tail`. The function `set_head` takes two arguments, the first of which must be a pair. It modifies this pair, replacing the head pointer by a pointer to the second argument of `set_head`.¹⁴

As an example, suppose that `x` is bound to the list `list(list("a", "b"), "c")` and `y` to the list `list("e", "f")` as illustrated in figure 3.12. Evaluating the expression `set_head(x, y)` modifies the pair to which `x` is bound, replacing its head by the value of `y`. The result of the operation is shown in figure 3.13. The structure `x` has been modified and would now be

¹⁴The functions `set_head` and `set_tail` return the value `undefined`. Like assignment, they should be used only for their effect.

printed as `list(list("e", "f"), "c", "d")`. The pairs representing the list `list("a", "b")`, identified by the pointer that was replaced, are now detached from the original structure.¹⁵

Compare figure 3.13 with figure 3.14, which illustrates the result of executing

```
const z = pair(y, tail(x));
```

with `x` and `y` bound to the original lists of figure 3.12. The variable `z` is now bound to a new pair created by the `pair` operation; the list to which `x` is bound is unchanged.

The `set_tail` operation is similar to `set_head`. The only difference is that the `tail` pointer of the pair, rather than the `head` pointer, is replaced. The effect of executing `set_tail(x, y)` on the lists of figure 3.12 is shown in figure 3.15. Here the `tail` pointer of `x` has been replaced by the pointer to `list("e", "f")`. Also, the list `list("c", "d")`, which used to be the tail of `x`, is now detached from the structure.

The function `pair` builds new list structure by creating new pairs, while `set_head` and `set_tail` modify existing pairs. Indeed, we could implement `pair` in terms of the two mutators, together with a function `get_new_pair`, which returns a new pair that is not part of any existing list structure. We obtain the new pair, set its head and tail pointers to the designated objects, and return the new pair as the result of the `pair`.¹⁶

```
function pair(x, y) {
  const fresh = get_new_pair();
  set_head(fresh, x);
  set_tail(fresh, y);
  return fresh;
}
```

Exercise 3.12

The following function for appending lists was introduced in section 2.2.1:

```
function append(x, y) {
  return is_null(x)
    ? y
    : pair(head(x), append(tail(x), y));
}
```

The function `append` forms a new list by successively pairing the elements of `x` onto `y`. The function `append_mutator` is similar to `append`, but it is a mutator rather than a constructor. It appends the lists by splicing them together, modifying the final pair of `x` so that its `tail` is now `y`. (It is an error to call `append_mutator` with an empty `x`.)

¹⁵We see from this that mutation operations on lists can create ‘garbage’ that is not part of any accessible structure.

¹⁶The function `get_new_pair` is one of the operations that must be implemented as part of the memory management required by a JavaScript implementation.

```
function append_mutator(x, y) {
    set_tail(last_pair(x), y);
}
```

Here `last_pair` is a function that returns the last pair in its argument:

```
function last_pair(x) {
    return is_null(tail(x))
        ? x
        : last_pair(tail(x));
}
```

Consider the program

```
const x = list("a", "b");
const y = list("c", "d");
const z = append(x, y);
display(z);           // ["a", ["b", ["c", ["d", null]]]]
display(tail(x)); // ???
const w = append_mutator(x, y);
display(w);           // ["a", ["b", ["c", ["d", null]]]]
display(tail(x)); // ???
```

What are the missing responses? Draw box-and-pointer diagrams to explain your answer.

Exercise 3.13

Consider the following `make_cycle` function, which uses the `last_pair` function defined in exercise 3.12:

```
function make_cycle(x) {
    set_tail(last_pair(x), x);
    return x;
}
```

Draw a box-and-pointer diagram that shows the structure `z` created by

```
const z = make_cycle(list("a", "b", "c"));
```

What happens if we try to compute `last_pair(z)`?

Exercise 3.14

The following function is quite useful, although obscure:

```
function mystery(x) {
  function loop(x, y) {
    if (is_null(x)) {
      return y;
    } else {
      let temp = tail(x);
      set_tail(x, y);
      return loop(temp, x);
    }
  }
  return loop(x, null);
}
```

The function `loop` uses the ‘temporary’ variable `temp` to hold the old value of the `tail` of `x`, since the `set_tail` on the next line destroys the `tail`. Explain what `mystery` does in general. Suppose `v` is defined by

```
const v = list("a", "b", "c");
```

Draw the box-and-pointer diagram that represents the list to which `v` is bound. Suppose that we now evaluate

```
const w = mystery(v);
```

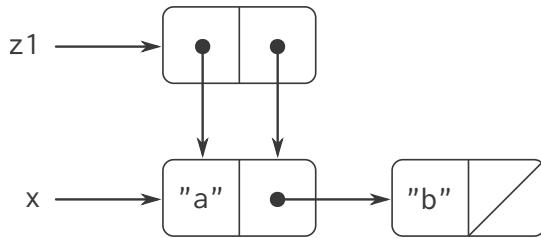
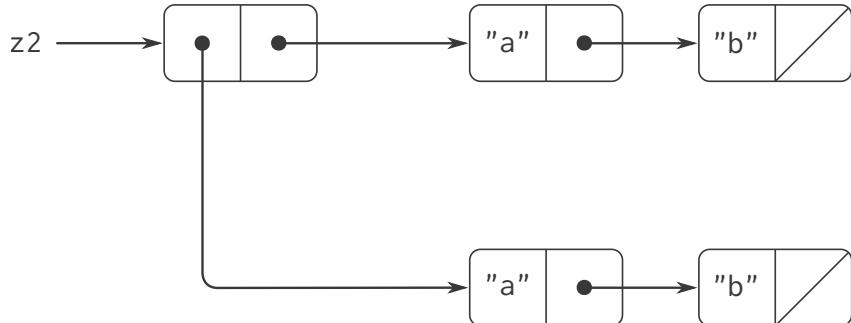
Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating this program. What would be printed as the values of `v` and `w`?

Sharing and identity

We mentioned in section 3.1.3 the theoretical issues of ‘sameness’ and ‘change’ raised by the introduction of assignment. These issues arise in practice when individual pairs are *shared* among different data objects. For example, consider the structure formed by

```
const x = list("a", "b");
const z1 = pair(x, x);
```

As shown in figure 3.16, `z1` is a pair whose head and tail both point to the same pair `x`. This sharing of `x` by the head and tail of `z1` is a consequence of the straightforward way in which `pair` is implemented. In general, using `pair` to construct lists will result in an interlinked structure of pairs in which many individual pairs are shared by many different structures.

Figure 3.16: The list z_1 formed by $\text{pair}(x, x)$.Figure 3.17: The list z_2 formed by $\text{pair}(\text{list}("a", "b"), \text{list}("a", "b"))$.

In contrast to figure 3.16, figure 3.17 shows the structure created by

```
const z2 = pair(list("a", "b"), list("a", "b));
```

In this structure, the pairs in the two $\text{list}("a", "b")$ lists are distinct, although they contain the same strings.¹⁷

When thought of as a list, z_1 and z_2 both represent ‘the same’ list:

```
list(list("a", "b"), "a", "b")
```

In general, sharing is completely undetectable if we operate on lists using only `pair`, `head`, and `tail`. However, if we allow mutators on list structure, sharing becomes significant. As an example of the difference that sharing can make, consider the following function, which modifies the head of the structure to which it is applied:

```
function set_to_wow(x) {
    set_head(head(x), "wow");
    return x;
}
```

Even though z_1 and z_2 are ‘the same’ structure, applying `set_to_wow` to them yields different results. With z_1 , altering the head also changes the tail, because in z_1 the head and the tail

¹⁷The two pairs are distinct because each call to `pair` returns a new pair. The strings are ‘the same’ in the sense that they are primitive data (just like numbers) that are composed of the same characters in the same order. Since JavaScript provides no way to mutate a string, any sharing that the designers of a JavaScript interpreter might decide to implement for strings is undetectable. We consider primitive data such as numbers, booleans and strings as *identical*, if and only if they are *indistinguishable*.

are the same pair. With `z2`, the head and `tail` are distinct, so `set_to_wow!` modifies only the head:

```

z1;
// displays: [[["a", ["b", null]], ["a", ["b", null]]]

set_to_wow(z1);
// displays: [[["wow", ["b", null]], ["wow", ["b", null]]]

z2;
// displays: [[["a", ["b", null]], ["a", ["b", null]]]

set_to_wow(z2);
// displays: [[["wow", ["b", null]], ["a", ["b", null]]]

```

One way to detect sharing in list structures is to use the predicate operator `>==`, which we introduced in section 2.3.1 as a way to test whether two strings are equal. More generally, `x == y` tests whether `x` and `y` are the same object (that is, whether `x` and `y` are equal as pointers). Thus, with `z1` and `z2` as defined in figures 3.16 and 3.17, `head(z1) == tail(z1)` is true and `head(z2) == tail(z2)` is false.

As will be seen in the following sections, we can exploit sharing to greatly extend the repertoire of data structures that can be represented by pairs. On the other hand, sharing can also be dangerous, since modifications made to structures will also affect other structures that happen to share the modified parts. The mutation operations `set_head` and `set_tail` should be used with care; unless we have a good understanding of how our data objects are shared, mutation can have unanticipated results.¹⁸

Exercise 3.15

Draw box-and-pointer diagrams to explain the effect of `set_to_wow` on the structures `z1` and `z2` above.

Exercise 3.16

Ben Bitdiddle decides to write a function to count the number of pairs in any list structure. ‘It’s easy,’ he reasons. ‘The number of pairs in any structure is the number in the head plus the number in the `tail` plus one more to count the current pair.’ So Ben writes the following

¹⁸The subtleties of dealing with sharing of mutable data objects reflect the underlying issues of ‘sameness’ and ‘change’ that were raised in section 3.1.3. We mentioned there that admitting change to our language requires that a compound object must have an ‘identity’ that is something different from the pieces from which it is composed. In JavaScript, we consider this ‘identity’ to be the quality that is tested by `==`, i.e., by equality of pointers. Since in most JavaScript implementations a pointer is essentially a memory address, we are ‘solving the problem’ of defining the identity of objects by stipulating that a data object ‘itself’ is the information stored in some particular set of memory locations in the computer. This suffices for simple JavaScript programs, but is hardly a general way to resolve the issue of ‘sameness’ in computational models.

function:

```
function count_pairs(x) {
    return !is_pair(x)
        ? 0
        : count_pairs(head(x)) +
            count_pairs(tail(x)) + 1;
}
```

Show that this function is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's function would return 3; return 4; return 7; never return at all.

Exercise 3.17

Devise a correct version of the `count_pairs` function of exercise 3.16 that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

Exercise 3.18

Write a function that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by taking successive tails would go into an infinite loop. Exercise 3.13 constructed such lists.

Exercise 3.19

Redo exercise 3.18 using an algorithm that takes only a constant amount of space. (This requires a very clever idea.)

Mutation is just assignment

When we introduced compound data, we observed in section 2.1.3 that pairs can be represented purely in terms of functions:

```
function pair(x, y) {
    function dispatch(m) {
        if (m === "head") {
            return x;
        } else if (m === "tail") {
            return y;
        } else {
            return "undefined operation -- pair";
        }
    }
}
```

```

    }
    return dispatch;
}

```

The same observation is true for mutable data. We can implement mutable data objects as functions using assignment and local state. For instance, we can extend the above pair implementation to handle `set_head` and `set_tail` in a manner analogous to the way we implemented bank accounts using `make-account` in section 3.1.1:

```

function pair(x, y) {
    function set_x(v) {
        x = v;
    }
    function set_y(v) {
        y = v;
    }
    return function dispatch(m) {
        if (m === "head") {
            return x;
        } else if (m === "tail") {
            return y;
        } else if (m === "set_head") {
            return set_x;
        } else if (m === "set_tail") {
            return set_y;
        } else {
            return "undefined operation - - pair";
        }
    };
}

function head(z) {
    return z("head");
}

function tail(z) {
    return z("tail");
}

function set_head(z, new_value) {
    (z("set_head"))(new_value);
    return z;
}

function set_tail(z, new_value) {
    (z("set_tail"))(new_value);
    return z;
}

```

Assignment is all that is needed, theoretically, to account for the behavior of mutable data. As soon as we admit assignment to our language, we raise all the issues, not only of assignment, but of mutable data in general.¹⁹

Exercise 3.20

Draw environment diagrams to illustrate the evaluation of the sequence of expressions

```
const x = pair(1, 2);
const z = pair(x, x);
set_head(tail(z), 17);
head(x);
```

using the functional implementation of pairs given above. (Compare exercise 3.11.)

3.3.2 Representing Queues

The mutators `set_head` and `set_tail` enable us to use pairs to construct data structures that cannot be built with `pair`, `head`, and `tail` alone. This section shows how to use pairs to represent a data structure called a queue. Section 3.3.3 will show how to represent data structures called tables.

A *queue* is a sequence in which items are inserted at one end (called the *rear* of the queue) and deleted from the other end (the *front*). Figure 3.18 shows an initially empty queue in which the items *a* and *b* are inserted. Then *a* is removed, *c* and *d* are inserted, and *b* is removed. Because items are always removed in the order in which they are inserted, a queue is sometimes called a *FIFO* (first in, first out) buffer.

<u>Operation</u>	<u>Resulting Queue</u>
<code>var q = make_queue();</code>	
<code>insert_queue(q, "a");</code>	<i>a</i>
<code>insert_queue(q, "b");</code>	<i>a b</i>
<code>delete_queue(q);</code>	<i>b</i>
<code>insert_queue(q, "c");</code>	<i>b c</i>
<code>insert_queue(q, "d");</code>	<i>b c d</i>
<code>delete_queue(q);</code>	<i>c d</i>

Figure 3.18: Queue operations.

In terms of data abstraction, we can regard a queue as defined by the following set of operations:

¹⁹On the other hand, from the viewpoint of implementation, assignment requires us to modify the environment, which is itself a mutable data structure. Thus, assignment and mutation are equipotent: Each can be implemented in terms of the other.

- a constructor:

`make_queue()` returns an empty queue (a queue containing no items).

- two selectors:

`empty_queue(queue)` tests if the queue is empty.

`front_queue(queue)` returns the object at the front of the queue, signaling an error if the queue is empty; it does not modify the queue.

- two mutators:

`insert_queue(queue, item)` inserts the item at the rear of the queue and returns the modified queue as its value.

`delete_queue(queue)` removes the item at the front of the queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

Because a queue is a sequence of items, we could certainly represent it as an ordinary list; the front of the queue would be the head of the list, inserting an item in the queue would amount to appending a new element at the end of the list, and deleting an item from the queue would just be taking the tail of the list. However, this representation is inefficient, because in order to insert an item we must scan the list until we reach the end. Since the only method we have for scanning a list is by successive tail operations, this scanning requires $\Theta(n)$ steps for a list of n items. A simple modification to the list representation overcomes this disadvantage by allowing the queue operations to be implemented so that they require $\Theta(1)$ steps; that is, so that the number of steps needed is independent of the length of the queue.

The difficulty with the list representation arises from the need to scan to find the end of the list. The reason we need to scan is that, although the standard way of representing a list as a chain of pairs readily provides us with a pointer to the beginning of the list, it gives us no easily accessible pointer to the end. The modification that avoids the drawback is to represent the queue as a list, together with an additional pointer that indicates the final pair in the list. That way, when we go to insert an item, we can consult the rear pointer and so avoid scanning the list.

A queue is represented, then, as a pair of pointers, `front_ptr` and `rear_ptr`, which indicate, respectively, the first and last pairs in an ordinary list. Since we would like the queue to be an identifiable object, we can use `pair` to combine the two pointers. Thus, the queue itself will be the pair of the two pointers. Figure 3.19 illustrates this representation.

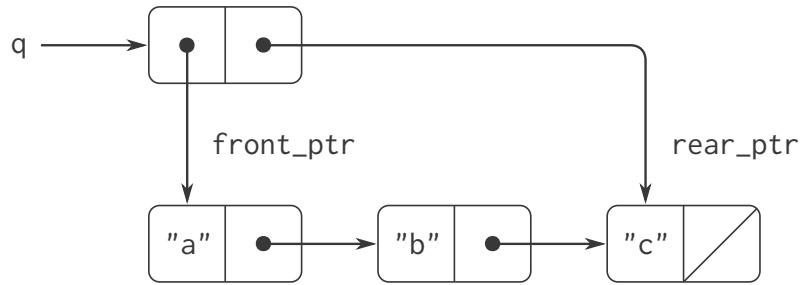


Figure 3.19: Implementation of a queue as a list with front and rear pointers.

To define the queue operations we use the following functions, which enable us to select and to modify the front and rear pointers of a queue:

```

function front_ptr(queue) {
  return head(queue);
}
function rear_ptr(queue) {
  return tail(queue);
}
function set_front_ptr(queue, item) {
  set_head(queue, item);
}
function set_rear_ptr(queue, item) {
  set_tail(queue, item);
}
  
```

Now we can implement the actual queue operations. We will consider a queue to be empty if its front pointer is the empty list:

```

function is_empty_queue(queue) {
  return is_null(front_ptr(queue));
}
  
```

The `make_queue` constructor returns, as an initially empty queue, a pair whose head and tail are both the empty list:

```

function make_queue() {
  return pair(null, null);
}
  
```

To select the item at the front of the queue, we return the head of the pair indicated by the front pointer:

```

function front_queue(queue) {
  return is_empty_queue(queue)
    ? Error("front_queue called with an empty queue",
           queue)
    : front_ptr(queue);
}
  
```

To insert an item in a queue, we follow the method whose result is indicated in Figure 3.20. We first create a new pair whose head is the item to be inserted and whose tail is the empty list. If the queue was initially empty, we set the front and rear pointers of the queue to this new pair. Otherwise, we modify the final pair in the queue to point to the new pair, and also set the rear pointer to the new pair.

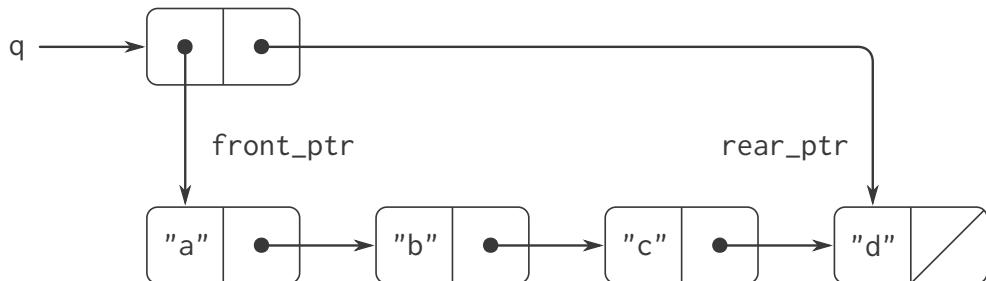


Figure 3.20: Result of using `insert_queue(q, 'd')` on the queue of Figure 3.19.

```
function insert_queue(queue, item) {
    const new_pair = pair(item, null);
    if (is_empty_queue(queue)) {
        set_front_ptr(queue, new_pair);
        set_rear_ptr(queue, new_pair);
    } else {
        set_tail(rear_ptr(queue), new_pair);
        set_rear_ptr(queue, new_pair);
    }
    return queue;
}
```

To delete the item at the front of the queue, we merely modify the front pointer so that it now points at the second item in the queue, which can be found by following the tail pointer of the first item (see Figure 3.21):²⁰

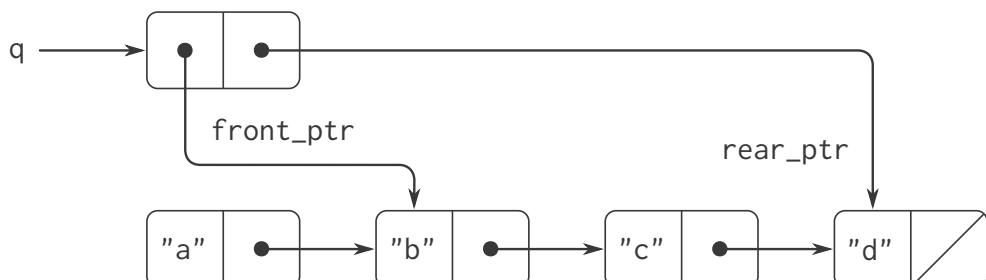


Figure 3.21: Result of using `delete_queue(q)` on the queue of Figure 3.20.

²⁰If the first item is the final item in the queue, the front pointer will be the empty list after the deletion, which will mark the queue as empty; we needn't worry about updating the rear pointer, which will still point to the deleted item, because `empty-queue?` looks only at the front pointer.

```
function delete_queue(queue) {
  if (is_empty_queue(queue)) {
    Error("delete_queue called with an empty queue",
          queue);
  } else {
    set_front_ptr(queue, tail(front_ptr(queue)));
    return queue;
  }
}
```

Exercise 3.21

Ben Bitdiddle decides to test the queue implementation described above. He types in the functions to the JavaScript interpreter and proceeds to try them out:

```
const q1 = make_queue();

insert_queue(q1, "a");

insert_queue(q1, "b");

delete_queue(q1);

delete_queue(q1);
```

'It's all wrong!' he complains. 'The interpreter's response shows that the last item is inserted into the queue twice. And when I delete both items, the second b is still there, so the queue isn't empty, even though it's supposed to be.' Eva Lu Ator suggests that Ben has misunderstood what is happening. 'It's not that the items are going into the queue twice,' she explains. 'It's just that the standard JavaScript printer doesn't know how to make sense of the queue representation. If you want to see the queue printed correctly, you'll have to define your own print function for queues.' Explain what Eva Lu is talking about. In particular, show why Ben's examples produce the printed results that they do. Define a function `print_queue` that takes a queue as input and prints the sequence of items in the queue.

Exercise 3.22

Instead of representing a queue as a pair of pointers, we can build a queue as a function with local state. The local state will consist of pointers to the beginning and the end of an ordinary list. Thus, the `make_queue` function will have the form

```
function make_queue() {
  function front_ptr(...) ...
  function rear_ptr(...) ...
  //definitions of internal functions
```

```

function dispatch(m) ...
  return dispatch;
}

```

Complete the definition of `make_queue` and provide implementations of the queue operations using this representation.

Exercise 3.23

A *deque* ('double-ended queue') is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make_deque`, the predicate `is_empty_deque`, selectors `front_deque` `front-deque` and `rear_deque` and mutators `front_insert_deque`, `front_delete_deque`, `rear_insert_deque`, and `rear_delete_deque`. Show how to represent deques using pairs, and give implementations of the operations.²¹ All operations should be accomplished in $\Theta(1)$ steps.

3.3.3 Representing Tables

When we studied various ways of representing sets in chapter 2, we mentioned in section 2.3.3 the task of maintaining a table of records indexed by identifying keys. In the implementation of data-directed programming in section 2.4.3, we made extensive use of two-dimensional tables, in which information is stored and retrieved using two keys. Here we see how to build tables as mutable list structures.

We first consider a one-dimensional table, in which each value is stored under a single key. We implement the table as a list of records, each of which is implemented as a pair consisting of a key and the associated value. The records are glued together to form a list by pairs whose heads point to successive records. These gluing pairs are called the *backbone* of the table. In order to have a place that we can change when we add a new record to the table, we build the table as a *headed list*. A headed list has a special backbone pair at the beginning, which holds a dummy 'record'—in this case the arbitrarily chosen string "`*table*`". Figure 3.22 shows the box-and-pointer diagram for the table

```

a: 1
b: 2
c: 3

```

²¹Be careful not to make the interpreter try to print a structure that contains cycles. (See exercise 3.13.)

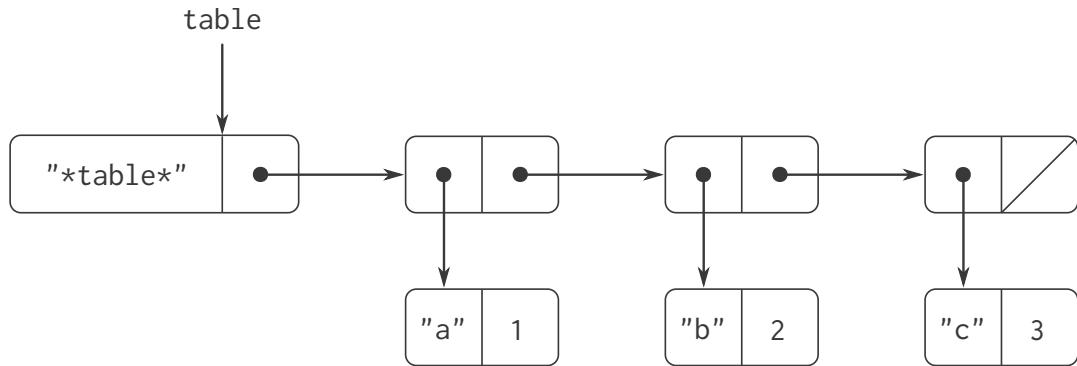


Figure 3.22: A table represented as a headed list.

To extract information from a table we use the `lookup` function, which takes a key as argument and returns the associated value (or false if there is no value stored under that key). The function `lookup` is defined in terms of the `assoc` operation, which expects a key and a list of records as arguments. Note that `assoc` never sees the dummy record. The function `assoc` returns the record that has the given key as its head.²² The function `lookup` then checks to see that the resulting record returned by `assoc` is not false, and returns the value (the tail) of the record.

```

function lookup(key, table) {
  const record = assoc(key, tail(table));
  return record === undefined
    ? undefined
    : tail(record);
}

function assoc(key, records) {
  return is_null(records)
    ? undefined
    : is_equal(key, head(head(records)))
      ? head(records)
      : assoc(key, tail(records));
}
  
```

To insert a value in a table under a specified key, we first use `assoc` to see if there is already a record in the table with this key. If not, we form a new record by pairing the key with the value, and insert this at the head of the table's list of records, after the dummy record. If there already is a record with this key, we set the tail of this record to the designated new value. The header of the table provides us with a fixed location to modify in order to insert the new record.²³

²²Because `assoc` uses `is_equal`, it can recognize keys that are strings, numbers, or list structure.

²³Thus, the first backbone pair is the object that represents the table ‘itself’; that is, a pointer to the table is a pointer to this pair. This same backbone pair always starts the table. If we did not arrange things in this way, `insert` would have to return a new value for the start of the table when it added a new record.

```
function insert(key, value, table) {
  const record = assoc(key, tail(table));
  return record === undefined
    ? set_tail(table, pair(pair(key, value),
                           tail(table)))
    : set_tail(record, value);
}
```

To construct a new table, we simply create a list containing the symbol `*table*`:

```
function make_table() {
  return list("*table*");
}
```

Two-dimensional tables

In a two-dimensional table, each value is indexed by two keys. We can construct such a table as a one-dimensional table in which each key identifies a subtable. Figure 3.23 shows the box-and-pointer diagram for the table

```
math:
+ 43
- 45
* 42
letters:
a 97
b 98
```

which has two subtables. (The subtables don't need a special header symbol, since the key that identifies the subtable serves this purpose.)

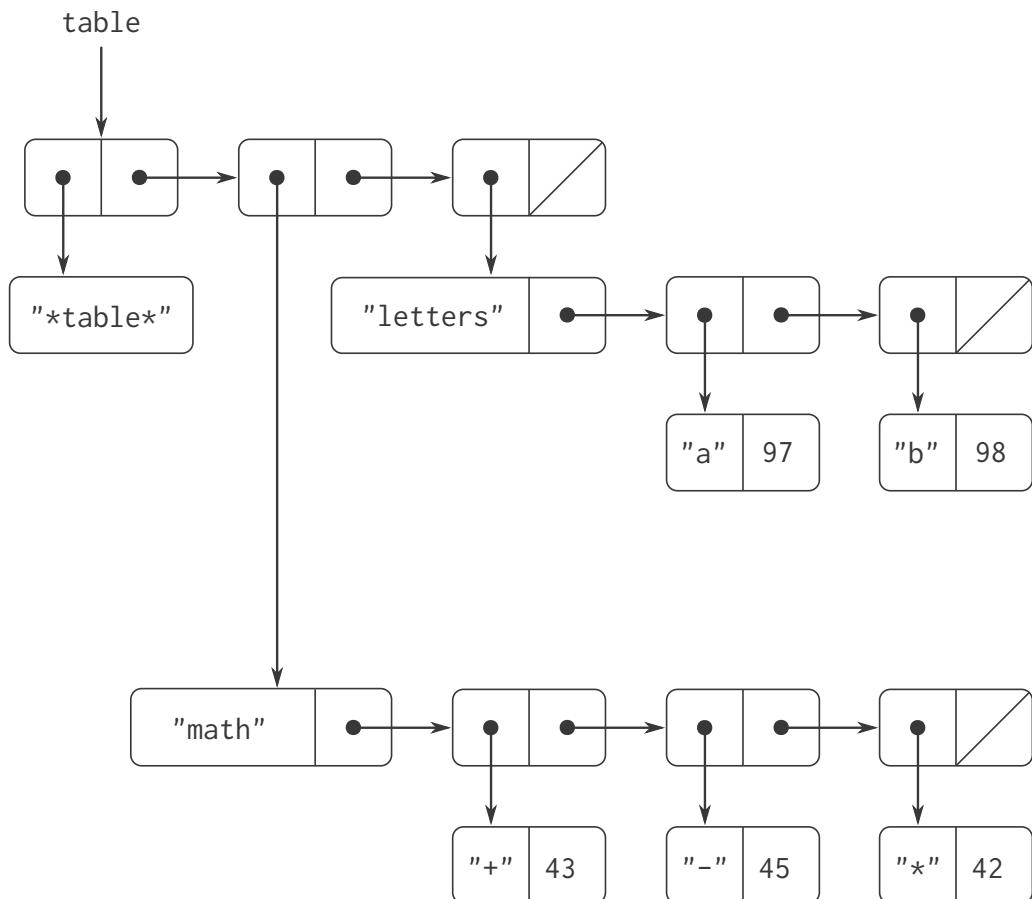


Figure 3.23: A two-dimensional table.

When we look up an item, we use the first key to identify the correct subtable. Then we use the second key to identify the record within the subtable.

```

function lookup(key_1, key_2, table) {
  const subtable = assoc(key_1, tail(table));
  if (subtable === undefined) {
    return undefined;
  } else {
    const record = assoc(key_2, tail(subtable));
    if (record === undefined) {
      return undefined;
    } else {
      return tail(record);
    }
  }
}
  
```

To insert a new item under a pair of keys, we use `assoc` to see if there is a subtable stored under the first key. If not, we build a new subtable containing the single record (`key_2, value`) and insert it into the table under the first key. If a subtable already exists for the first key, we insert the new record into this subtable, using the insertion method for one-dimensional tables

described above:

```
function insert(key_1, key_2, value, table) {
    const subtable = assoc(key_1, tail(table));
    if (subtable === undefined) {
        set_tail(table,
                  pair(list(key_1, pair(key_2, value)),
                        tail(table)));
    } else {
        const record = assoc(key_2, tail(table));
        if (record === undefined) {
            set_tail(subtable,
                      pair(pair(key_2, value),
                            tail(subtable)));
        } else {
            set_tail(record, value);
        }
    }
}
```

Creating local tables

The lookup and insert operations defined above take the table as an argument. This enables us to use programs that access more than one table. Another way to deal with multiple tables is to have separate lookup and insert functions for each table. We can do this by representing a table procedurally, as an object that maintains an internal table as part of its local state. When sent an appropriate message, this ‘table object’ supplies the function with which to operate on the internal table. Here is a generator for two-dimensional tables represented in this fashion:

```
function make_table() {
    const local_table = list("*table*");
    function lookup(key_1, key_2) {
        const subtable = assoc(key_1, tail(local_table));
        if (subtable === undefined) {
            return undefined;
        } else {
            const record = assoc(key_2, tail(subtable));
            if (record === undefined) {
                return undefined;
            } else {
                return tail(record);
            }
        }
    }
    function insert(key_1, key_2, value) {
        const subtable = assoc(key_1, tail(local_table));
        if (subtable === undefined) {
            set_tail(local_table,
                      pair(list(key_1, pair(key_2, value)),
                            tail(local_table)));
        } else {
            const record = assoc(key_2, tail(subtable));
            if (record === undefined) {
                set_tail(subtable,
                          pair(pair(key_2, value),
                                tail(subtable)));
            } else {
                set_tail(record, value);
            }
        }
    }
}
```

```

if (subtable === undefined) {
    set_tail(local_table,
              pair(list(key_1, pair(key_2, value)),
                    tail(local_table)));
} else {
    const record = assoc(key_2, tail(subtable));
    if (record === undefined) {
        set_tail(subtable,
                  pair(pair(key_2, value),
                        tail(subtable)));
    } else {
        set_tail(record, value);
    }
}
function dispatch(m) {
    return m === "lookup"
        ? lookup
        : m === "insert"
        ? insert
        : "undefined operation -- table";
}
return dispatch;
}

```

Using `make_table`, we could implement the get and put operations used in section 2.4.3 for data-directed programming, as follows:

```

const operation_table = make_table();
const get = operation_table("lookup");
const put = operation_table("insert");

```

The function `get` takes as arguments two keys, and `put` takes as arguments two keys and a value. Both operations access the same local table, which is encapsulated within the object created by the call to `make_table`.

Exercise 3.24

In the table implementations above, the keys are tested for equality using `is_equal` (called by `assoc`). This is not always the appropriate test. For instance, we might have a table with numeric keys in which we don't need an exact match to the number we're looking up, but only a number within some tolerance of it. Design a table constructor `make_table` that takes as an argument a `same_key` function that will be used to test 'equality' of keys. The function `make_table` should return a `dispatch` function that can be used to access appropriate `lookup` and `insert` functions for a local table.

Exercise 3.25

Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The `lookup` and `insert` functions should take as input a list of keys used to access the table.

Exercise 3.26

To search a table as implemented above, one needs to scan through the list of records. This is basically the unordered list representation of section 2.3.3. For large tables, it may be more efficient to structure the table in a different manner. Describe a table implementation where the (key, value) records are organized using a binary tree, assuming that keys can be ordered in some way (e.g., numerically or alphabetically). (Compare exercise 2.66 of chapter 2.)

Exercise 3.27

Memoization (also called *tabulation*) is a technique that enables a function to record, in a local table, values that have previously been computed. This technique can make a vast difference in the performance of a program. A memoized function maintains a table in which values of previous calls are stored using as keys the arguments that produced the values. When the memoized function is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table. As an example of memoization, recall from section 1.2.2 the exponential process for computing Fibonacci numbers:

```
function fib(n) {
    return n === 0
        ? 0
        : n === 1
        ? 1
        : fib(n - 1) + fib(n - 2);
}
```

The memoized version of the same function is

```
const memo_fib = memoize(n => n === 0
    ? 0
    : n === 1
    ? 1
    : memo_fib(n - 1) +
      memo_fib(n - 2)
);
```

where the memoizer is defined as

```
function memoize(f) {
    const table = make_table();
    return x => {
        const previously_computed_result
            = lookup(x, table);
        if (previously_computed_result === undefined) {
            const result = f(x);
            insert(x, result, table);
            return result;
        } else {
            return previously_computed_result;
        }
    };
}
```

Draw an environment diagram to analyze the computation of `memo_fib(3)`. Explain why `memo_fib` computes the n th Fibonacci number in a number of steps proportional to n . Would the scheme still work if we had simply defined `memo_fib` to be `memoize(fib)`?

3.3.4 A Simulator for Digital Circuits

Designing complex digital systems, such as computers, is an important engineering activity. Digital systems are constructed by interconnecting simple elements. Although the behavior of these individual elements is simple, networks of them can have very complex behavior. Computer simulation of proposed circuit designs is an important tool used by digital systems engineers. In this section we design a system for performing digital logic simulations. This system typifies a kind of program called an *event-driven simulation*, in which actions ('events') trigger further events that happen at a later time, which in turn trigger more events, and so so.

Our computational model of a circuit will be composed of objects that correspond to the elementary components from which the circuit is constructed. There are *wires*, which carry *digital signals*. A digital signal may at any moment have only one of two possible values, 0 and 1. There are also various types of digital *function boxes*, which connect wires carrying input signals to other output wires. Such boxes produce output signals computed from their input signals. The output signal is delayed by a time that depends on the type of the function box. For example, an *inverter* is a primitive function box that inverts its input. If the input signal to an inverter changes to 0, then one *inverter-delay* later the inverter will change its output signal to 1. If the input signal to an inverter changes to 1, then one *inverter-delay* later the inverter will change its output signal to 0. We draw an inverter symbolically as in Figure 3.24. An *and-gate*, also shown in Figure 3.24, is a primitive function box with two inputs and one output. It drives its output signal to a value that is the *logical and* of the inputs. That is, if both

of its input signals become 1, then one *and-gate-delay* time later the and-gate will force its output signal to be 1; otherwise the output will be 0. An *or-gate* is a similar two-input primitive function box that drives its output signal to a value that is the *logical or* of the inputs. That is, the output will become 1 if at least one of the input signals is 1; otherwise the output will become 0.

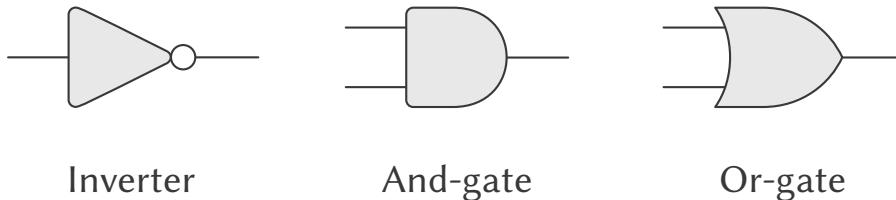


Figure 3.24: Primitive functions in the digital logic simulator.

We can connect primitive functions together to construct more complex functions. To accomplish this we wire the outputs of some function boxes to the inputs of other function boxes. For example, the *half-adder* circuit shown in Figure 3.25 consists of an or-gate, two and-gates, and an inverter. It takes two input signals, A and B, and has two output signals, S and C. S will become 1 whenever precisely one of A and B is 1, and C will become 1 whenever A and B are both 1. We can see from the figure that, because of the delays involved, the outputs may be generated at different times. Many of the difficulties in the design of digital circuits arise from this fact.

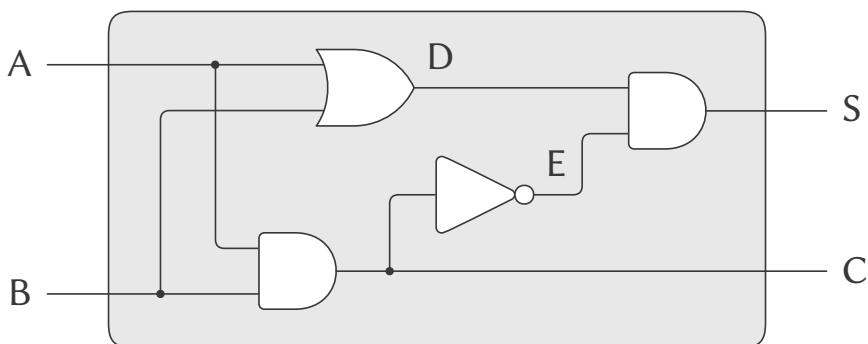


Figure 3.25: A half-adder circuit.

We will now build a program for modeling the digital logic circuits we wish to study. The program will construct computational objects modeling the wires, which will ‘hold’ the signals. Function boxes will be modeled by functions that enforce the correct relationships among the signals.

One basic element of our simulation will be a function `make_wire`, which constructs wires. For example, we can construct six wires as follows:

```
const a = make_wire();
const b = make_wire();
```

```
const c = make_wire();
const d = make_wire();
const e = make_wire();
const s = make_wire();
```

We attach a function box to a set of wires by calling a function that constructs that kind of box. The arguments to the constructor function are the wires to be attached to the box. For example, given that we can construct and-gates, or-gates, and inverters, we can wire together the half-adder shown in Figure 3.25:

```
or_gate(a, b, d);
and_gate(a, b, c);
inverter(c, e);
and_gate(d, e, s);
```

Better yet, we can explicitly name this operation by defining a function `half_adder` that constructs this circuit, given the four external wires to be attached to the half-adder:

```
function half_adder(a, b, s, c) {
  const d = make_wire();
  const e = make_wire();
  or_gate(a, b, d);
  and_gate(a, b, c);
  inverter(c, e);
  and_gate(d, e, s);
  return "ok";
}
```

The advantage of making this definition is that we can use `half_adder` itself as a building block in creating more complex circuits. Figure 3.26, for example, shows a *full-adder* composed of two half-adders and an or-gate.²⁴ We can construct a full-adder as follows:

```
function full_adder(a, b, c_in, sum, c_out) {
  const s = make_wire();
  const c1 = make_wire();
  const c2 = make_wire();
  half_adder(b, c_in, s, c1);
  half_adder(a, s, sum, c2);
  or_gate(c1, c2, c_out);
  return "ok";
}
```

²⁴A full-adder is a basic circuit element used in adding two binary numbers. Here A and B are the bits at corresponding positions in the two numbers to be added, and C_{in} is the carry bit from the addition one place to the right. The circuit generates SUM, which is the sum bit in the corresponding position, and C_{out} , which is the carry bit to be propagated to the left.

Having defined `full_adder` as a function, we can now use it as a building block for creating still more complex circuits. (For example, see exercise 3.30.)

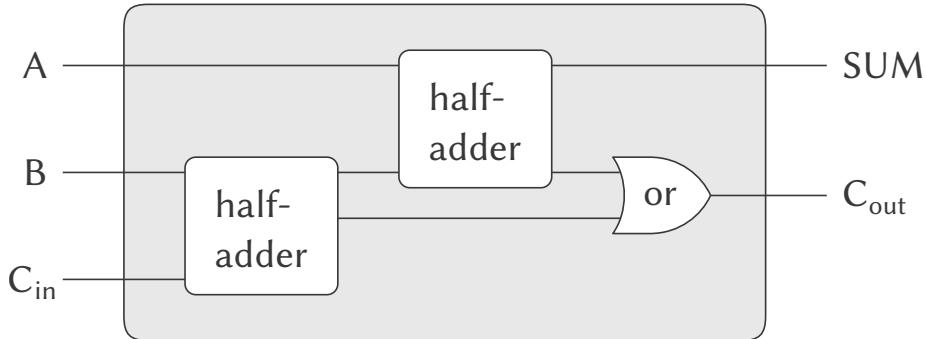


Figure 3.26: A full-adder circuit.

In essence, our simulator provides us with the tools to construct a language of circuits. If we adopt the general perspective on languages with which we approached the study of JavaScript in section 1.1, we can say that the primitive function boxes form the primitive elements of the language, that wiring boxes together provides a means of combination, and that specifying wiring patterns as functions serves as a means of abstraction.

Primitive function boxes

The primitive function boxes implement the ‘forces’ by which a change in the signal on one wire influences the signals on other wires. To build function boxes, we use the following operations on wires:

- `get_signal(wire)`: returns the current value of the signal on the wire.
- `set_signal(wire, new_value)`: changes the value of the signal on the wire to the new value.
- `add_action(wire, nullary_function)`: asserts that the designated function should be run whenever the signal on the wire changes value. Such functions are the vehicles by which changes in the signal value on the wire are communicated to other wires.

In addition, we will make use of a function `after_delay` that takes a time delay and a function to be run and executes the given function after the given delay.

Using these functions, we can define the primitive digital logic functions. To connect an input to an output through an inverter, we use `add_action` to associate with the input wire a function that will be run whenever the signal on the input wire changes value. The function computes the `logical_not` of the input signal, and then, after one `inverter_delay`, sets the output signal to be this new value:

```

function inverter(input, output) {
    function invert_input() {
        const new_value = logical_not(get_signal(input));
        after_delay(inverter_delay,
                    () => set_signal(output, new_value));
    }
    add_action(input, invert_input);
    return "ok";
}

function logical_not(s) {
    return s === 0
        ? 1
        : s === 1
        ? 0
        : Error("Invalid signal for logical_not", s);
}

```

An and-gate is a little more complex. The action function must be run if either of the inputs to the gate changes. It computes the logical_and (using a function analogous to logical_not) of the values of the signals on the input wires and sets up a change to the new value to occur on the output wire after one and_gate_delay.

```

function and_gate(a1, a2, output) {
    function and_action_function() {
        const new_value = logical_and(get_signal(a1),
                                      get_signal(a2));
        after_delay(and_gate_delay,
                    () => set_signal(output, new_value));
    }
    add_action(a1, and_action_function);
    add_action(a2, and_action_function);
    return "ok";
}

```

Exercise 3.28

Define an or-gate as a primitive function box. Your or_gate constructor should be similar to and_gate.

Exercise 3.29

Another way to construct an or-gate is as a compound digital logic device, built from and-gates and inverters. Define a function or_gate that accomplishes this. What is the delay time of the or-gate in terms of and_gate_delay and inverter_delay?

Exercise 3.30

Figure 3.27 shows a *ripple-carry adder* formed by stringing together n full-adders. This is the simplest form of parallel adder for adding two n -bit binary numbers. The inputs $A_1, A_2, A_3, \dots, A_n$ and $B_1, B_2, B_3, \dots, B_n$ are the two binary numbers to be added (each A_k and B_k is a 0 or a 1). The circuit generates $S_1, S_2, S_3, \dots, S_n$, the n bits of the sum, and C , the carry from the addition. Write a function `ripple_carry_adder` that generates this circuit. The function should take as arguments three lists of n wires each—the A_k , the B_k , and the S_k —and also another wire C . The major drawback of the ripple-carry adder is the need to wait for the carry signals to propagate. What is the delay needed to obtain the complete output from an n -bit ripple-carry adder, expressed in terms of the delays for and-gates, or-gates, and inverters?

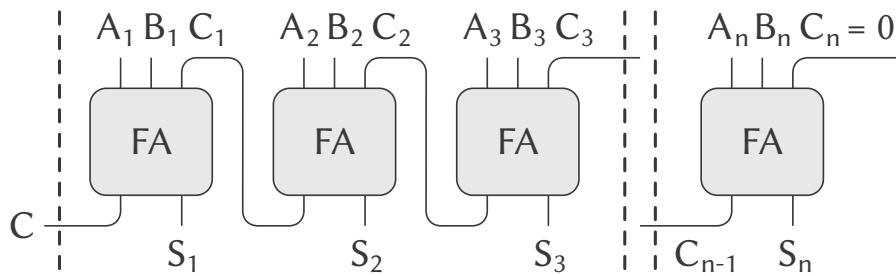


Figure 3.27: A ripple-carry adder for n -bit numbers.

Representing wires

A wire in our simulation will be a computational object with two local state variables: a `signal_value` (initially taken to be 0) and a collection of `action_function` to be run when the signal changes value. We implement the wire, using message-passing style, as a collection of local functions together with a `dispatch` function that selects the appropriate local operation, just as we did with the simple bank-account object in section 3.1.1:

```
function make_wire() {
    let signal_value = 0;
    let action_functions = null;
    function set_my_signal(new_value) {
        if (signal_value !== new_value) {
            signal_value = new_value;
            call_each(action_functions);
        } else {}
    }
    function accept_action_function(fun) {
        action_functions = pair(fun, action_functions);
        fun();
    }
    function dispatch(m) {
```

```

        return m === "get_signal"
            ? signal_value
            : m === "set_signal"
            ? set_my_signal
            : m === "add_action"
            ? accept_action_function
            : Error("Unknown operation in wire", m);
    }
    return dispatch;
}

```

The local function `set_my_signal` tests whether the new signal value changes the signal on the wire. If so, it runs each of the action functions, using the following function `call_each`, which calls each of the items in a list of no-argument functions:

```

function call_each(functions) {
    if (is_null(functions)) {
        return "done";
    } else {
        (head(functions))();
        call_each(tail(functions));
    }
}

```

The local function `accept_action_function` adds the given function to the list of functions to be run, and then runs the new function once. (See exercise 3.31.)

With the local `dispatch` function set up as specified, we can provide the following functions to access the local operations on wires:²⁵

```

function get_signal(wire) {
    return wire("get_signal");
}

function set_signal(wire, new_value) {
    return (wire("set_signal"))(new_value);
}

function add_action(wire, action_function) {
    return (wire("add_action"))(action_function);
}

```

²⁵These functions are simply syntactic sugar that allow us to use ordinary functional syntax to access the local functions of objects. It is striking that we can interchange the role of ‘functions’ and ‘data’ in such a simple way. For example, if we write `wire('get_signal')` we think of `wire` as a function that is called with the message “`get_signal`” as input. Alternatively, writing `get_signal(wire)` encourages us to think of `wire` as a data object that is the input to a function `get_signal`. The truth of the matter is that, in a language in which we can deal with functions as objects, there is no fundamental difference between ‘functions’ and ‘data,’ and we can choose our syntactic sugar to allow us to program in whatever style we choose.

Wires, which have time-varying signals and may be incrementally attached to devices, are typical of mutable objects. We have modeled them as functions with local state variables that are modified by assignment. When a new wire is created, a new set of state variables is allocated (by the `let` expression in `make_wire`) and a new dispatch function is constructed and returned, capturing the environment with the new state variables.

The wires are shared among the various devices that have been connected to them. Thus, a change made by an interaction with one device will affect all the other devices attached to the wire. The wire communicates the change to its neighbors by calling the action functions provided to it when the connections were established.

The agenda

The only thing needed to complete the simulator is `after_delay`. The idea here is that we maintain a data structure, called an *agenda*, that contains a schedule of things to do. The following operations are defined for agendas:

- `make_agenda()`: returns a new empty agenda.
- `is_empty_agenda(agenda)`: is true if the specified agenda is empty.
- `first_agenda_item(agenda)`: returns the first item on the agenda.
- `remove_first_agenda_item(agenda)`: modifies the agenda by removing the first item.
- `add_to_agenda(time, action, agenda)`: modifies the agenda by adding the given action function to be run at the specified time.
- `current_time(agenda)`: returns the current simulation time.

The particular agenda that we use is denoted by `the_agenda`. The function `after_delay` adds new elements to `the_agenda`:

```
function after_delay(delay, action) {
    add_to_agenda(delay + current_time(the_agenda),
                  action, the_agenda);
}
```

The simulation is driven by the function `propagate`, which operates on `the_agenda`, executing each function on the agenda in sequence. In general, as the simulation runs, new items will be added to the agenda, and `propagate` will continue the simulation as long as there are items on the agenda:

```
function propagate() {
    if (is_empty_agenda(the_agenda)) {
```

```

    return "done";
} else {
    const first = first_agenda_item(the_agenda);
    first();
    remove_first_agenda_item(the_agenda);
    return propagate();
}
}

```

A sample simulation

The following function, which places a ‘probe’ on a wire, shows the simulator in action. The probe tells the wire that, whenever its signal changes value, it should print the new signal value, together with the current time and a name that identifies the wire:

```

function probe(name, wire) {
    add_action(wire,
        () => display(name + " " +
                        current_time(the_agenda) +
                        ", New value = " +
                        get_signal(wire));
}

```

We begin by initializing the agenda and specifying delays for the primitive function boxes:

```

const the_agenda = make_agenda();
const inverter_delay = 2;
const and_gate_delay = 3;
const or_gate_delay = 5;

```

Now we define four wires, placing probes on two of them:

```

const input_1 = make_wire();
const input_2 = make_wire();
const sum = make_wire();
const carry = make_wire();

probe("Sum", sum);
// Sum 0, New value = 0

probe("Carry", carry);
// Carry 0, New value = 0

```

Next we connect the wires in a half-adder circuit (as in Figure 3.25), set the signal on `input_1` to 1, and run the simulation:

```

half_adder(input_1, input_2, sum, carry);

set_signal(input_1, 1);

```

```
propagate();
// Sum 8, New Value = 1
```

The sum signal changes to 1 at time 8. We are now eight time units from the beginning of the simulation. At this point, we can set the signal on input_2 to 1 and allow the values to propagate:

```
set_signal(input_2, 1);

propagate();
// Carry 11, New value = 1
// Sum 16, New value = 0
```

The carry changes to 1 at time 11 and the sum changes to 0 at time 16.

Exercise 3.31

The internal function accept_action_function defined in make_wire specifies that when a new action function is added to a wire, the function is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the system's response would differ if we had defined accept_action_function as

```
function accept_action_function(fun) {
    action_functions = pair(fun, action_functions);
}
```

Implementing the agenda

Finally, we give details of the agenda data structure, which holds the functions that are scheduled for future execution.

The agenda is made up of *time segments*. Each time segment is a pair consisting of a number (the time) and a queue (see exercise 3.32) that holds the functions that are scheduled to be run during that time segment.

```
function make_time_segment(time, queue) {
    return pair(time, queue);
}

function segment_time(s) {
    return head(s);
}

function segment_queue(s) {
    return tail(s);
}
```

We will operate on the time-segment queues using the queue operations described in section 3.3.2.

The agenda itself is a one-dimensional table of time segments. It differs from the tables described in section 3.3.3 in that the segments will be sorted in order of increasing time. In addition, we store the *current time* (i.e., the time of the last action that was processed) at the head of the agenda. A newly constructed agenda has no time segments and has a current time of 0:²⁶

```
function make_agenda() {
    return list(0);
}

function current_time(agenda) {
    return head(agenda);
}

function set_current_time(agenda, time) {
    set_head(agenda, time);
}

function segments(agenda) {
    return tail(agenda);
}

function set_segments(agenda, segs) {
    set_tail(agenda, segs);
}

function first_segment(agenda) {
    return head(segments(agenda));
}

function rest_segments(agenda) {
    return tail(segments(agenda));
}
```

An agenda is empty if it has no time segments:

```
function is_empty_agenda(agenda) {
    return is_null(segments(agenda));
}
```

To add an action to an agenda, we first check if the agenda is empty. If so, we create a time segment for the action and install this in the agenda. Otherwise, we scan the agenda, examining the time of each segment. If we find a segment for our appointed time, we add the action to the

²⁶The agenda is a headed list, like the tables in section 3.3.3, but since the list is headed by the time, we do not need an additional dummy header (such as the *table* symbol used with tables).

associated queue. If we reach a time later than the one to which we are appointed, we insert a new time segment into the agenda just before it. If we reach the end of the agenda, we must create a new time segment at the end.

```
function add_to_agenda(time, action, agenda) {
  function belongs_before(segs) {
    return is_null(segs) ||
           time < segment_time(head(segs));
  }
  function make_new_time_segment(time, action) {
    const q = make_queue();
    insert_queue(q, action);
    return make_time_segment(time, q);
  }
  function add_to_segments(segs) {
    if (segment_time(head(segs)) === time) {
      insert_queue(segment_queue(head(segs)), action);
    } else {
      const rest = tail(segs);
      if (belongs_before(rest)) {
        set_tail(segs,
                  pair(make_new_time_segment(time, action),
                        tail(segs)));
      } else {
        add_to_segments(rest);
      }
    }
  }
  const segs = segments(agenda);
  if (belongs_before(segs)) {
    set_segments(agenda,
                 pair(make_new_time_segment(time, action),
                       segs));
  } else {
    add_to_segments(segs);
  }
}
```

The function that removes the first item from the agenda deletes the item at the front of the queue in the first time segment. If this deletion makes the time segment empty, we remove it from the list of segments:²⁷

```
function remove_first_agenda_item(agenda) {
  const q = segment_queue(first_segment(agenda));
  delete_queue(q);
```

²⁷Observe that the **if** expression in this function has no alternative expression. Such a ‘one-armed **if** statement’ is used to decide whether to do something, rather than to select between two expressions. An **if** expression returns an unspecified value if the predicate is false and there is no alternative.

```

if (is_empty_queue(q)) {
    set_segments(agenda, rest_segments(agenda));
} else {}
}

```

The first agenda item is found at the head of the queue in the first time segment. Whenever we extract an item, we also update the current time.²⁸

```

function first_agenda_item(agenda) {
    if (is_empty_agenda(agenda)) {
        error("Agenda is empty: first_agenda_item");
    } else {
        const first_seg = first_segment(agenda);
        set_current_time(agenda, segment_time(first_seg));
        return front_queue(segment_queue(first_seg));
    }
}

```

Exercise 3.32

The functions to be run during each time segment of the agenda are kept in a queue. Thus, the functions for each segment are called in the order in which they were added to the agenda (first in, first out). Explain why this order must be used. In particular, trace the behavior of an and-gate whose inputs change from 0,1 to 1,0 in the same segment and say how the behavior would differ if we stored a segment's functions in an ordinary list, adding and removing functions only at the front (last in, first out).

3.3.5 Propagation of Constraints

Computer programs are traditionally organized as one-directional computations, which perform operations on prespecified arguments to produce desired outputs. On the other hand, we often model systems in terms of relations among quantities. For example, a mathematical model of a mechanical structure might include the information that the deflection d of a metal rod is related to the force F on the rod, the length L of the rod, the cross-sectional area A , and the elastic modulus E via the equation

$$dAE = FL$$

Such an equation is not one-directional. Given any four of the quantities, we can use it to compute the fifth. Yet translating the equation into a traditional computer language would

²⁸In this way, the current time will always be the time of the action most recently processed. Storing this time at the head of the agenda ensures that it will still be available even if the associated time segment has been deleted.

force us to choose one of the quantities to be computed in terms of the other four. Thus, a function for computing the area A could not be used to compute the deflection d , even though the computations of A and d arise from the same equation.²⁹

In this section, we sketch the design of a language that enables us to work in terms of relations themselves. The primitive elements of the language are *primitive constraints*, which state that certain relations hold between quantities. For example, `adder(a, b, c)` specifies that the quantities a , b , and c must be related by the equation $a + b = c$, `multiplier(x, y, z)` expresses the constraint $xy = z$, and `constant(3.14, x)` says that the value of x must be 3.14.

Our language provides a means of combining primitive constraints in order to express more complex relations. We combine constraints by constructing *constraint networks*, in which constraints are joined by *connectors*. A connector is an object that ‘holds’ a value that may participate in one or more constraints. For example, we know that the relationship between Fahrenheit and Celsius temperatures is

$$9C = 5(F - 32)$$

Such a constraint can be thought of as a network consisting of primitive adder, multiplier, and constant constraints (figure 3.28). In the figure, we see on the left a multiplier box with three terminals, labeled m_1 , m_2 , and p . These connect the multiplier to the rest of the network as follows: The m_1 terminal is linked to a connector C , which will hold the Celsius temperature. The m_2 terminal is linked to a connector w , which is also linked to a constant box that holds 9. The p terminal, which the multiplier box constrains to be the product of m_1 and m_2 , is linked to the p terminal of another multiplier box, whose m_2 is connected to a constant 5 and whose m_1 is connected to one of the terms in a sum.

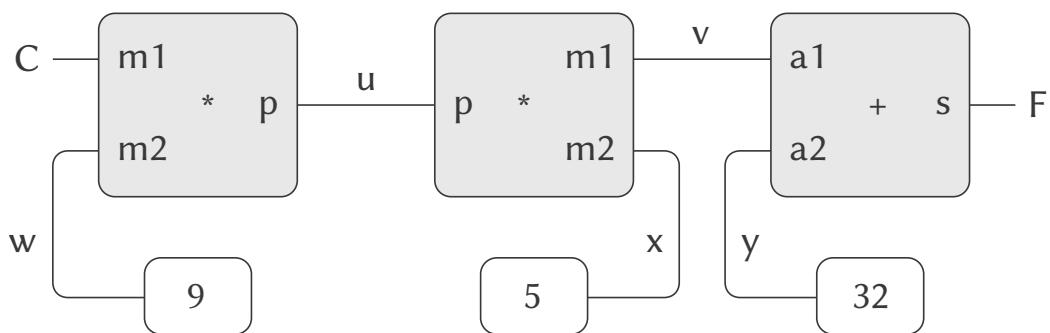


Figure 3.28: The relation $9C = 5(F - 32)$ expressed as a constraint network.

Computation by such a network proceeds as follows: When a connector is given a value (by

²⁹Constraint propagation first appeared in the incredibly forward-looking SKETCHPAD system of Ivan Sutherland (1963). A beautiful constraint-propagation system based on the Smalltalk language was developed by Alan Borning (1977) at Xerox Palo Alto Research Center. Sussman, Stallman, and Steele applied constraint propagation to electrical circuit analysis (Sussman and Stallman 1975; Sussman and Steele 1980). TK!Solver (Konopasek and Jayaraman 1984) is an extensive modeling environment based on constraints.

the user or by a constraint box to which it is linked), it awakens all of its associated constraints (except for the constraint that just awakened it) to inform them that it has a value. Each awakened constraint box then polls its connectors to see if there is enough information to determine a value for a connector. If so, the box sets that connector, which then awakens all of its associated constraints, and so on. For instance, in conversion between Celsius and Fahrenheit, w , x , and y are immediately set by the constant boxes to 9, 5, and 32, respectively. The connectors awaken the multipliers and the adder, which determine that there is not enough information to proceed. If the user (or some other part of the network) sets C to a value (say 25), the leftmost multiplier will be awakened, and it will set u to $25 \cdot 9 = 225$. Then u awakens the second multiplier, which sets v to 45, and v awakens the adder, which sets F to 77.

Using the constraint system

To use the constraint system to carry out the temperature computation outlined above, we first create two connectors, C and F , by calling the constructor `make_connector`, and link C and F in an appropriate network:

```
const C = make_connector();
const F = make_connector();
celsius_fahrenheit_converter(C, F);
```

The function that creates the network is defined as follows:

```
function celsius_fahrenheit_converter(c, f) {
    const u = make_connector();
    const v = make_connector();
    const w = make_connector();
    const x = make_connector();
    const y = make_connector();
    multiplier(c, w, u);
    multiplier(v, x, u);
    adder(v, y, f);
    constant(9, w);
    constant(5, x);
    constant(32, y);
    return "ok";
}
```

This function creates the internal connectors u , v , w , x , and y , and links them as shown in Figure 3.28 using the primitive constraint constructors `adder`, `multiplier`, and `constant`. Just as with the digital-circuit simulator of section 3.3.4, expressing these combinations of primitive elements in terms of functions automatically provides our language with a means of abstraction for compound objects.

To watch the network in action, we can place probes on the connectors C and F , using a probe

function similar to the one we used to monitor wires in section 3.3.4. Placing a probe on a connector will cause a message to be printed whenever the connector is given a value:

```
probe("Celsius Temp", C);
probe("Fahrenheit Temp", F);
```

Next we set the value of C to 25. (The third argument to `set_value` tells C that this directive comes from the user.)

```
set_value(C, 25, "user");
// Probe: Celsius Temp = 25
// Probe: Fahrenheit Temp = 77
```

The probe on C awakens and reports the value. C also propagates its value through the network as described above. This sets F to 77, which is reported by the probe on F.

Now we can try to set F to a new value, say 212:

```
set_value(F, 212, "user");
// Error! Contradiction (77 212)
```

The connector complains that it has sensed a contradiction: Its value is 77, and someone is trying to set it to 212. If we really want to reuse the network with new values, we can tell C to forget its old value:

```
forget_value(C, "user");
// Probe: Celsius Temp = ?
// Probe: Fahrenheit Temp = ?
```

C finds that the "user", who set its value originally, is now retracting that value, so C agrees to lose its value, as shown by the probe, and informs the rest of the network of this fact. This information eventually propagates to F, which now finds that it has no reason for continuing to believe that its own value is 77. Thus, F also gives up its value, as shown by the probe.

Now that F has no value, we are free to set it to 212:

```
set_value(F, 212, "user");
// Probe: Fahrenheit Temp = 212
// Probe: Celsius Temp = 100
```

This new value, when propagated through the network, forces C to have a value of 100, and this is registered by the probe on C. Notice that the very same network is being used to compute C given F and to compute F given C. This nondirectionality of computation is the distinguishing feature of constraint-based systems.

Implementing the constraint system

The constraint system is implemented via procedural objects with local state, in a manner very similar to the digital-circuit simulator of section 3.3.4. Although the primitive objects of the constraint system are somewhat more complex, the overall system is simpler, since there is no concern about agendas and logic delays.

The basic operations on connectors are the following:

- `has_value(connector)`: tells whether the connector has a value.
- `get_value(connector)`: returns the connector's current value.
- `set_value(connector, new_value, informant)`: indicates that the informant is requesting the connector to set its value to the new value.
- `forget_value(connector, retractor)`: tells the connector that the retractor is requesting it to forget its value.
- `connect(connector, new_constraint)`: tells the connector to participate in the new constraint.

The connectors communicate with the constraints by means of the functions `inform_about_value`, which tells the given constraint that the connector has a value, and `forget_value`, which tells the constraint that the connector has lost its value.

Adder constructs an adder constraint among summand connectors `a1` and `a2` and a `sum` connector. An adder is implemented as a function with local state (the function `me` below):

```
function adder(a1, a2, sum) {
    function process_new_value() {
        if (has_value(a1) && has_value(a2)) {
            set_value(sum, get_value(a1) + get_value(a2), me);
        } else if (has_value(a1) && has_value(sum)) {
            set_value(a2, get_value(sum) - get_value(a1), me);
        } else if (has_value(a2) && has_value(sum)) {
            set_value(a1, get_value(sum) - get_value(a2), me);
        } else {
        }
    }
    function process_forget_value() {
        forget_value(sum, me);
        forget_value(a1, me);
        forget_value(a2, me);
        process_new_value();
    }
    function me(request) {
```

```

    if (request === "I-have-a-value") {
        process_new_value();
    } else if (request === "I-lost-my-value") {
        process_forget_value();
    } else {
        Error("Unknown request in adder", request);
    }
}
connect(a1, me);
connect(a2, me);
connect(sum, me);
return me;
}

```

Adder connects the new adder to the designated connectors and returns it as its value. The function `me`, which represents the adder, acts as a dispatch to the local functions. The following ‘syntax interfaces’ (see footnote 25 in section 3.3.4) are used in conjunction with the dispatch:

```

function inform_about_value(constraint) {
    return constraint("I-have-a-value");
}

function inform_about_no_value(constraint) {
    return constraint("I-lost-my-value");
}

```

The adder’s local function `process_new_value` is called when the adder is informed that one of its connectors has a value. The adder first checks to see if both `a1` and `a2` have values. If so, it tells `sum` to set its value to the sum of the two addends. The `informant` argument to `set_value` is `me`, which is the adder object itself. If `a1` and `a2` do not both have values, then the adder checks to see if perhaps `a1` and `sum` have values. If so, it sets `a2` to the difference of these two. Finally, if `a2` and `sum` have values, this gives the adder enough information to set `a1`. If the adder is told that one of its connectors has lost a value, it requests that all of its connectors now lose their values. (Only those values that were set by this adder are actually lost.) Then it runs `process_new_value`. The reason for this last step is that one or more connectors may still have a value (that is, a connector may have had a value that was not originally set by the adder), and these values may need to be propagated back through the adder.

A multiplier is very similar to an adder. It will set its product to 0 if either of the factors is 0, even if the other factor is not known.

```

function multiplier(m1, m2, product) {
    function process_new_value() {
        if ((has_value(m1) && get_value(m1) === 0)
            || (has_value(m2) && get_value(m2) === 0)) {
            set_value(product, 0, me);
        } else if (has_value(m1) && has_value(m2)) {

```

```

        set_value(product,
                  get_value(m1) * get_value(m2),
                  me);
    } else if (has_value(product) && has_value(m1)) {
        set_value(m2,
                  get_value(product) / get_value(m1),
                  me);
    } else if (has_value(product) && has_value(m2)) {
        set_value(m1,
                  get_value(product) / get_value(m2),
                  me);
    } else {
    }
}
function process_forget_value() {
    forget_value(product, me);
    forget_value(m1, me);
    forget_value(m2, me);
    process_new_value();
}
function me(request) {
    if (request === "I-have-a-value") {
        process_new_value();
    } else if (request === "I-lost-my-value") {
        process_forget_value();
    } else {
        Error("Unknown request in multiplier", request);
    }
}
connect(m1, me);
connect(m2, me);
connect(product, me);
return me;
}

```

A constant constructor simply sets the value of the designated connector. Any "I-have-a-value" or "I-lost-my-value" message sent to the constant box will produce an error.

```

function constant(value, connector) {
    function me(request) {
        Error("Unknown request in constant", request);
    }
    connect(connector, me);
    set_value(connector, value, me);
    return me;
}

```

Finally, a probe prints a message about the setting or unsetting of the designated connector:

```

function probe(name, connector) {
  function print_probe(value) {
    display("Probe: " + name + " = " + value);
  }
  function process_new_value() {
    print_probe(get_value(connector));
  }
  function process_forget_value() {
    print_probe("?");
  }
  function me(request) {
    return request === "I-have-a-value"
      ? process_new_value()
      : request === "I-lost-my-value"
        ? process_forget_value()
        : Error("Unknown request in probe",
          request);
  }
  connect(connector, me);
  return me;
}

```

Representing connectors

A connector is represented as a procedural object with local state variables `value`, the current value of the connector; `informant`, the object that set the connector's value; and `constraints`, a list of the constraints in which the connector participates.

```

function make_connector() {
  let value = false;
  let informant = false;
  let constraints = null;
  function set_my_value(newval, setter) {
    if (!has_value(me)) {
      value = newval;
      informant = setter;
      for_each_except(setter,
        inform_about_value,
        constraints);
    } else if (value !== newval) {
      error("Contradiction " +
        "(" + stringify(value) + ", " +
        + stringify(newval) + ")");
    } else {
      return "ignored";
    }
  }
}

```

```

function forget_my_value(retractor) {
    if (retractor === informant) {
        informant = false;
        for_each_except(retractor,
                        inform_about_no_value,
                        constraints);
    } else {
        return "ignored";
    }
}
function connect(new_constraint) {
    if (is_null(member(new_constraint,
                         constraints))) {
        constraints = pair(new_constraint, constraints);
    } else {
    }
    if (has_value(me)) {
        inform_about_value(new_constraint);
    } else {
    }

    return "done";
}
function me(request) {
    if (request === "has_value") {
        return informant !== false;
    } else if (request === "value") {
        return value;
    } else if (request === "set_value") {
        return set_my_value;
    } else if (request === "forget") {
        return forget_my_value;
    } else if (request === "connect") {
        return connect;
    } else {
        Error("Unknown operation in connector", request);
    }
}
return me;
}

```

The connector's local function `set_my_value` is called when there is a request to set the connector's value. If the connector does not currently have a value, it will set its value and remember as `informant` the constraint that requested the value to be set.³⁰ Then the connector will notify all of its participating constraints except the constraint that requested the value to be set. This is accomplished using the following iterator, which applies a designated function to

³⁰The setter might not be a constraint. In our temperature example, we used `user` as the setter.

all items in a list except a given one:

```
function for_each_except(exception, fun, list) {
  function loop(items) {
    if (is_null(items)) {
      return "done";
    } else if (head(items) === exception) {
      return loop(tail(items));
    } else {
      fun(head(items));
      return loop(tail(items));
    }
  }
  return loop(list);
}
```

If a connector is asked to forget its value, it runs the local function `forget_my_value`, which first checks to make sure that the request is coming from the same object that set the value originally. If so, the connector informs its associated constraints about the loss of the value.

The local function `connect` adds the designated new constraint to the list of constraints if it is not already in that list. Then, if the connector has a value, it informs the new constraint of this fact.

The connector's function `me` serves as a dispatch to the other internal functions and also represents the connector as an object. The following functions provide a syntax interface for the dispatch:

```
function has_value(connector) {
  return connector("has_value");
}

function get_value(connector) {
  return connector("value");
}

function set_value(connector, new_value, informant) {
  return (connector("set_value"))(new_value, informant);
}

function forget_value(connector, retractor) {
  return (connector("forget"))(retractor);
}

function connect(connector, new_constraint) {
  return (connector("connect"))(new_constraint);
}
```

Exercise 3.33

Using primitive multiplier, adder, and constant constraints, define a function `averager` that takes three connectors `a`, `b`, and `c` as inputs and establishes the constraint that the value of `c` is the average of the values of `a` and `b`.

Exercise 3.34

Louis Reasoner wants to build a squarer, a constraint device with two terminals such that the value of connector `b` on the second terminal will always be the square of the value `a` on the first terminal. He proposes the following simple device made from a multiplier:

```
function squarer(a, b) {
    return multiplier(a, a, b);
}
```

There is a serious flaw in this idea. Explain.

Exercise 3.35

Ben Bitdiddle tells Louis that one way to avoid the trouble in exercise 3.34 is to define a squarer as a new primitive constraint. Fill in the missing portions in Ben's outline for a function to implement such a constraint:

```
function squarer(a, b) {
    function process_new_value() {
        if (has_value(b)) {
            if (get_value(b) < 0) {
                Error("Square less than 0 in squarer",
                      get_value(b));
            } else {
                // alternative1...
            } else {
                // alternative2...
            }
        }
    }
    function process_forget_value() {
        // body1...
    }
    function me(request) {
        // body2...
    }
    // rest of definition
    return me;
}
```

Exercise 3.36

Suppose we evaluate the following sequence of expressions in the global environment:

```
const a = make_connector();
const b = make_connector();
set_value(a, 10, "user");
```

At some time during evaluation of the `set_value`, the following expression from the connector's local function is evaluated:

```
for_each_except(setter, inform_about_value, constraints);
```

Draw an environment diagram showing the environment in which the above expression is evaluated.

Exercise 3.37

The `celsius_fahrenheit_converter` function is cumbersome when compared with a more expression-oriented style of definition, such as

```
function celsius_fahrenheit_converter(x) {
    return cplus(cmul(cdiv(cv(9), cv(5)), x), cv(32));
}
```

Here `cplus`, `cmul`, etc. are the 'constraint' versions of the arithmetic operations. For example, `cplus` takes two connectors as arguments and returns a connector that is related to these by an adder constraint:

```
function cplus(x, y) {
    const z = make_connector();
    adder(x, y, z);
    return z;
}
```

Define analogous functions `cminus`, `cmul`, `cdiv`, and `cv` (constant value) that enable us to define compound constraints as in the converter example above.³¹

³¹The expression-oriented format is convenient because it avoids the need to name the intermediate expressions in a computation. Our original formulation of the constraint language is cumbersome in the same way that many languages are cumbersome when dealing with operations on compound data. For example, if we wanted to compute the product $(a + b) \cdot (c + d)$, where the variables represent vectors, we could work in 'imperative style,' using functions that set the values of designated vector arguments but do not themselves return vectors as values:

```
v_sum('a', 'b', temp1);
v_sum('c', 'd', temp2);
v_prod(temp1, temp2, answer);
```

Alternatively, we could deal with expressions, using functions that return vectors as values, and thus avoid explicitly mentioning `temp1` and `temp2`:

3.4 Concurrency: Time Is of the Essence

We've seen the power of computational objects with local state as tools for modeling. Yet, as Section 3.1.3 warned, this power extracts a price: the loss of referential transparency, giving rise to a thicket of questions about sameness and change, and the need to abandon the substitution model of evaluation in favor of the more intricate environment model.

The central issue lurking beneath the complexity of state, sameness, and change is that by introducing assignment we are forced to admit *time* into our computational models. Before we introduced assignment, all our programs were timeless, in the sense that any expression that has a value always has the same value. In contrast, recall the example of modeling withdrawals from a bank account and returning the resulting balance, introduced at the beginning of Section 3.1.1:

```
withdraw(25); // output: 75
withdraw(25); // output: 50
```

Here successive evaluations of the same expression yield different values. This behavior arises from the fact that the execution of assignment statements (in this case, assignments to the variable `balance`) delineates *moments in time* when values change. The result of evaluating an expression depends not only on the expression itself, but also on whether the evaluation occurs before or after these moments. Building models in terms of computational objects with local state forces us to confront time as an essential concept in programming.

We can go further in structuring computational models to match our perception of the physical world. Objects in the world do not change one at a time in sequence. Rather we perceive them as acting *concurrently*—all at once. So it is often natural to model systems as collections of computational processes that execute concurrently. Just as we can make our programs modular by organizing models in terms of objects with separate local state, it is often appropriate to divide computational models into parts that evolve separately and concurrently. Even if the programs are to be executed on a sequential computer, the practice of writing programs as if they were to be executed concurrently forces the programmer to avoid inessential timing

```
const answer = v_prod(v_sum('a', 'b'), v_sum('c', 'd'));
```

Since JavaScript allows us to return compound objects as values of functions, we can transform our imperative-style constraint language into an expression-oriented style as shown in this exercise. In languages that are impoverished in handling compound objects, such as Algol, Basic, and Pascal (unless one explicitly uses Pascal pointer variables), one is usually stuck with the imperative style when manipulating compound objects. Given the advantage of the expression-oriented format, one might ask if there is any reason to have implemented the system in imperative style, as we did in this section. One reason is that the non-expression-oriented constraint language provides a handle on constraint objects (e.g., the value of the `adder` function) as well as on connector objects. This is useful if we wish to extend the system with new operations that communicate with constraints directly rather than only indirectly via operations on connectors. Although it is easy to implement the expression-oriented style in terms of the imperative implementation, it is very difficult to do the converse.

constraints and thus makes programs more modular.

In addition to making programs more modular, concurrent computation can provide a speed advantage over sequential computation. Sequential computers execute only one operation at a time, so the amount of time it takes to perform a task is proportional to the total number of operations performed.³²

However, if it is possible to decompose a problem into pieces that are relatively independent and need to communicate only rarely, it may be possible to allocate pieces to separate computing processors, producing a speed advantage proportional to the number of processors available.

Unfortunately, the complexities introduced by assignment become even more problematic in the presence of concurrency. The fact of concurrent execution, either because the world operates in parallel or because our computers do, entails additional complexity in our understanding of time.

3.4.1 The Nature of Time in Concurrent Systems

On the surface, time seems straightforward. It is an ordering imposed on events.³³ For any events A and B , either A occurs before B , A and B are simultaneous, or A occurs after B . For instance, returning to the bank account example, suppose that Peter withdraws \$10 and Paul withdraws \$25 from a joint account that initially contains \$100, leaving \$65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either \$100 → \$90 → \$65\$100 → \$75 → \$65. In a computer implementation of the banking system, this changing sequence of balances could be modeled by successive assignments to a variable balance.

In complex situations, however, such a view can be problematic. Suppose that Peter and Paul, and other people besides, are accessing the same bank account through a network of banking machines distributed all over the world. The actual sequence of balances in the account will depend critically on the detailed timing of the accesses and the details of the communication among the machines.

This indeterminacy in the order of events can pose serious problems in the design of concurrent systems. For instance, suppose that the withdrawals made by Peter and Paul are implemented as two separate processes sharing a common variable balance, each process specified by the function given in Section 3.1.1:

³²Most real processors actually execute a few operations at a time, following a strategy called *pipelining*. Although this technique greatly improves the effective utilization of the hardware, it is used only to speed up the execution of a sequential instruction stream, while retaining the behavior of the sequential program.

³³To quote some graffiti seen on a Cambridge building wall: ‘Time is a device that was invented to keep everything from happening at once.’

```
function withdraw(amount) {
    if (balance >= amount) {
        balance = balance - amount;
        return balance;
    } else {
        return "Insufficient funds";
    }
}
```

If the two processes operate independently, then Peter might test the balance and attempt to withdraw a legitimate amount. However, Paul might withdraw some funds in between the time that Peter checks the balance and the time Peter completes the withdrawal, thus invalidating Peter's test.

Things can be worse still. Consider the expression

```
balance = balance - amount;
```

executed as part of each withdrawal process. This consists of three steps: (1) accessing the value of the balance variable; (2) computing the new balance; (3) setting balance to this new value. If Peter and Paul's withdrawals execute this statement concurrently, then the two withdrawals might interleave the order in which they access balance and set it to the new value.

The timing diagram in Figure 3.29 depicts an order of events where balance starts at 100, Peter withdraws 10, Paul withdraws 25, and yet the final value of balance is 75. As shown in the diagram, the reason for this anomaly is that Paul's assignment of 75 to balance is made under the assumption that the value of balance to be decremented is 100. That assumption, however, became invalid when Peter changed balance to 90. This is a catastrophic failure for the banking system, because the total amount of money in the system is not conserved. Before the transactions, the total amount of money was \$100. Afterwards, Peter has \$10, Paul has \$25, and the bank has \$75.³⁴

The general phenomenon illustrated here is that several processes may share a common state variable. What makes this complicated is that more than one process may be trying to manipulate the shared state at the same time. For the bank account example, during each transaction, each customer should be able to act as if the other customers did not exist. When a customer changes the balance in a way that depends on the balance, he must be able to assume that, just before the moment of change, the balance is still what he thought it was.

³⁴An even worse failure for this system could occur if the two assignment statements attempt to change the balance simultaneously, in which case the actual data appearing in memory might end up being a random combination of the information being written by the two processes. Most computers have interlocks on the primitive memory-write operations, which protect against such simultaneous access. Even this seemingly simple kind of protection, however, raises implementation challenges in the design of multiprocessor computers, where elaborate *cache-coherence* protocols are required to ensure that the various processors will maintain a consistent view of memory contents, despite the fact that data may be replicated ('cached') among the different processors to increase the speed of memory access.

JavaScript and concurrency

In its initial design, JavaScript did not allow for two processes to apply functions such as `withdraw` concurrently. In fact, the concurrency model of the language enforced strict sequential execution of activities resulting from *events*, with the use of an *event queue*. In the early 2000s, multicore computers became common and around 2010, the JavaScript designers responded with the introduction of concurrent processes via the concept of *web workers*. As of 2019, most internet browsers support this feature. As originally conceived, web workers were not able to share data such as the variable `balance` above. However, a shared data structure called `SharedArrayBuffer` is included in the latest [ECMAScript specification](#). Using `SharedArrayBuffer`, it is possible to program a `withdraw` function as described above.³⁵

Correct behavior of concurrent programs

The above example typifies the subtle bugs that can creep into concurrent programs. The root of this complexity lies in the assignments to variables that are shared among the different processes. We already know that we must be careful in writing programs that use assignment, because the results of a computation depend on the order in which the assignments occur.³⁶

With concurrent processes we must be especially careful about assignments, because we may not be able to control the order of the assignments made by the different processes. If several such changes might be made concurrently (as with two depositors accessing a joint account) we need some way to ensure that our system behaves correctly. For example, in the case of withdrawals from a joint bank account, we must ensure that money is conserved. To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.

³⁵As of 2019, browsers differ in their support for `SharedArrayBuffer` objects.

³⁶The factorial program in section 3.1.3 illustrates this for a single sequential process.

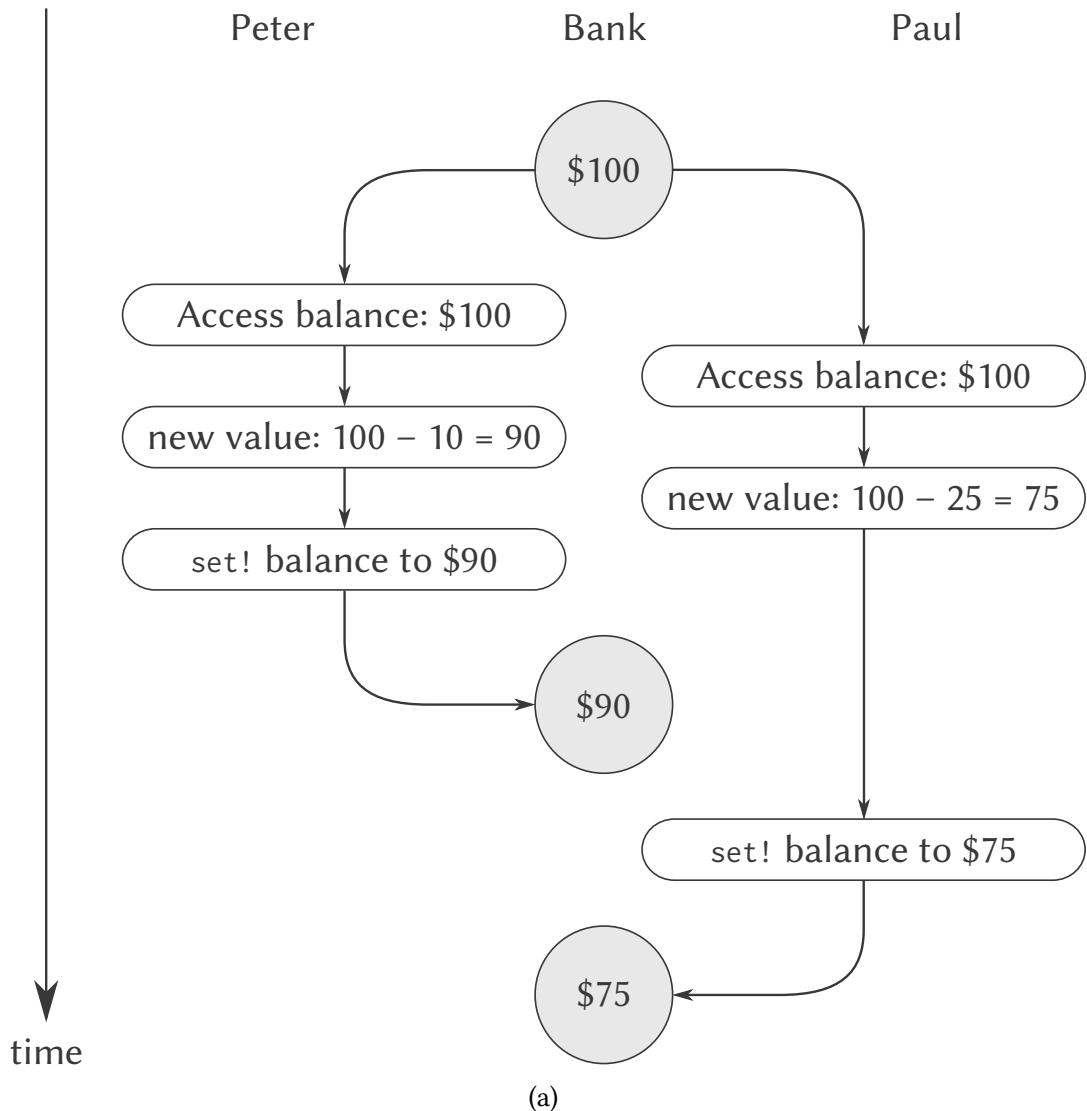


Figure 3.29: Timing diagram showing how interleaving the order of events in two banking withdrawals can lead to an incorrect final balance.

One possible restriction on concurrency would stipulate that no two operations that change any shared state variables can occur at the same time. This is an extremely stringent requirement. For distributed banking, it would require the system designer to ensure that only one transaction could proceed at a time. This would be both inefficient and overly conservative. Figure 3.30 shows Peter and Paul sharing a bank account, where Paul has a private account as well. The diagram illustrates two withdrawals from the shared account (one by Peter and one by Paul) and a deposit to Paul's private account.³⁷

The two withdrawals from the shared account must not be concurrent (since both access and update the same account), and Paul's deposit and withdrawal must not be concurrent (since

³⁷The columns show the contents of Peter's wallet, the joint account (in Bank1), Paul's wallet, and Paul's private account (in Bank2), before and after each withdrawal (W) and deposit (D). Peter withdraws \$10 from Bank1; Paul deposits \$5 in Bank2, then withdraws \$25 from Bank1.

both access and update the amount in Paul’s wallet). But there should be no problem permitting Paul’s deposit to his private account to proceed concurrently with Peter’s withdrawal from the shared account.

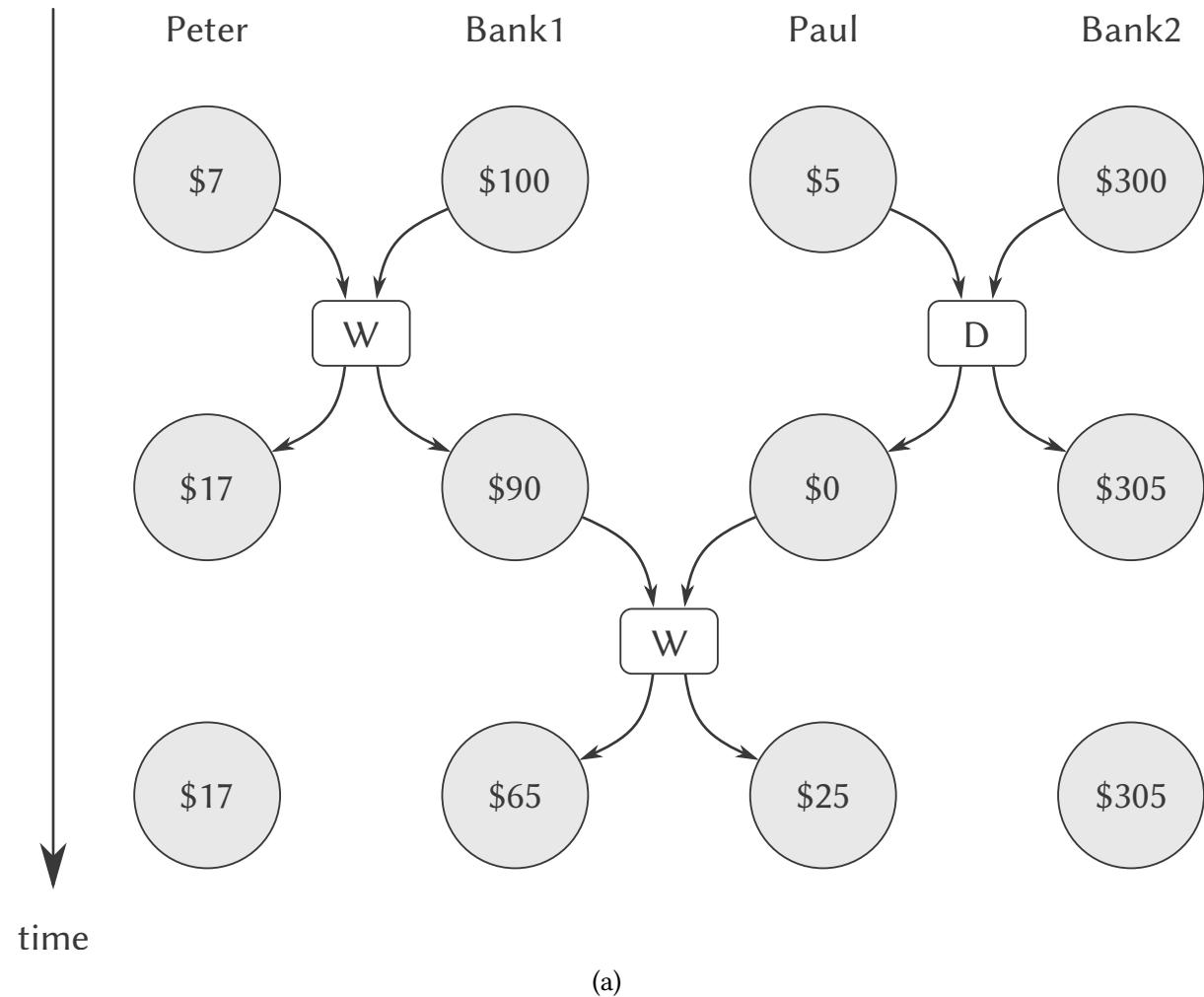


Figure 3.30: Concurrent deposits and withdrawals from a joint account in Bank1 and a private account in Bank2.

A less stringent restriction on concurrency would ensure that a concurrent system produces the same result as if the processes had run sequentially in some order. There are two important aspects to this requirement. First, it does not require the processes to actually run sequentially, but only to produce results that are the same *as if* they had run sequentially. For the example in Figure 3.30, the designer of the bank account system can safely allow Paul’s deposit and Peter’s withdrawal to happen concurrently, because the net result will be the same as if the two operations had happened sequentially. Second, there may be more than one possible ‘correct’ result produced by a concurrent program, because we require only that the result be the same as for *some* sequential order. For example, suppose that Peter and Paul’s joint account starts out with \$100, and Peter deposits \$40 while Paul concurrently withdraws half the money in the account. Then sequential execution could result in the account balance being either \$70

or \$90 (see exercise 3.38).³⁸

There are still weaker requirements for correct execution of concurrent programs. A program for simulating diffusion (say, the flow of heat in an object) might consist of a large number of processes, each one representing a small volume of space, that update their values concurrently. Each process repeatedly changes its value to the average of its own value and its neighbors' values. This algorithm converges to the right answer independent of the order in which the operations are done; there is no need for any restrictions on concurrent use of the shared values.

Exercise 3.38

Suppose that Peter, Paul, and Mary share a joint bank account that initially contains \$100. Concurrently, Peter deposits \$10, Paul withdraws \$20, and Mary withdraws half the money in the account, by executing the following commands:

- a. List all the different possible values for balance after these three transactions have been completed, assuming that the banking system forces the three processes to run sequentially in some order.
- b. What are some other values that could be produced if the system allows the processes to be interleaved? Draw timing diagrams like the one in figure 3.29 to explain how these values can occur.

3.4.2 Mechanisms for Controlling Concurrency

We've seen that the difficulty in dealing with concurrent processes is rooted in the need to consider the interleaving of the order of events in the different processes. For example, suppose we have two processes, one with three ordered events (a, b, c) and one with three ordered events (x, y, z) . If the two processes run concurrently, with no constraints on how their execution is interleaved, then there are 20 different possible orderings for the events that are consistent with the individual orderings for the two processes:

$$\begin{array}{cccc}
 (a, b, c, x, y, z) & (a, x, b, y, c, z) & (x, a, b, c, y, z) & (x, a, y, z, b, c) \\
 (a, b, x, c, y, z) & (a, x, b, y, z, c) & (x, a, b, y, c, z) & (x, y, a, b, c, z) \\
 (a, b, x, y, c, z) & (a, x, y, b, c, z) & (x, a, b, y, z, c) & (x, y, a, b, z, c) \\
 (a, b, x, y, z, c) & (a, x, y, b, z, c) & (x, a, y, b, c, z) & (x, y, a, z, b, c) \\
 (a, x, b, c, y, z) & (a, x, y, z, b, c) & (x, a, y, b, z, c) & (x, y, z, a, b, c)
 \end{array}$$

³⁸A more formal way to express this idea is to say that concurrent programs are inherently *nondeterministic*. That is, they are described not by single-valued functions, but by functions whose results are sets of possible values.

As programmers designing this system, we would have to consider the effects of each of these 20 orderings and check that each behavior is acceptable. Such an approach rapidly becomes unwieldy as the numbers of processes and events increase.

A more practical approach to the design of concurrent systems is to devise general mechanisms that allow us to constrain the interleaving of concurrent processes so that we can be sure that the program behavior is correct. Many mechanisms have been developed for this purpose. In this section, we describe one of them, the *serializer*.

Serializing access to shared state

Serialization implements the following idea: Processes will execute concurrently, but there will be certain collections of functions that cannot be executed concurrently. More precisely, serialization creates distinguished sets of functions such that only one execution of a function in each serialized set is permitted to happen at a time. If some function in the set is being executed, then a process that attempts to execute any function in the set will be forced to wait until the first execution has finished.

We can use serialization to control access to shared variables. For example, if we want to update a shared variable based on the previous value of that variable, we put the access to the previous value of the variable and the assignment of the new value to the variable in the same function. We then ensure that no other function that assigns to the variable can run concurrently with this function by serializing all of these functions with the same serializer. This guarantees that the value of the variable cannot be changed between an access and the corresponding assignment.

Serializers in JavaScript

To make the above mechanism more concrete, suppose that we have extended JavaScript to include a function called `parallel_execute`:

```
parallel_execute( f1, f2, ..., fk )
```

Each `f` must be a function of no arguments. The function `parallel_execute` creates a separate process for each `f`, which applies `f` (to no arguments). These processes all run concurrently.³⁹

As an example of how this is used, consider

```
let x = 10;

parallel_execute( () => { x = x * x; },
                  () => { x = x + 1; } );
```

³⁹The function `parallel_execute` is not part of the JavaScript standard, but it can be implemented using the `SharedArrayBuffer` feature mentioned in section 3.4.1.

This creates two concurrent processes— P_1 , which sets x to x times x , and P_2 , which increments x . After execution is complete, x will be left with one of five possible values, depending on the interleaving of the events of P_1 and P_2 :

- 101: P_1 sets x to 100 and then P_2 increments x to 101.
- 121: P_2 increments x to 11 and then P_1 sets x to x times x .
- 110: P_2 changes x from 10 to 11 between the two times that P_1 accesses the value of x during the evaluation of $x * x$.
- 11: P_2 accesses x , then P_1 sets x to 100, then P_2 sets x .
- 100: P_1 accesses x (twice), then P_2 sets x to 11, then P_1 sets x .

We can constrain the concurrency by using serialized functions, which are created by *serializers*. Serializers are constructed by `make_serializer`, whose implementation is given below. A serializer takes a function as argument and returns a serialized function that behaves like the original function. All calls to a given serializer return serialized functions in the same set.

Thus, in contrast to the example above, executing

```
let x = 10;

const s = make_serializer();

parallel_execute(s( () => { x = x * x; }),  

                 s( () => { x = x + 1; } ));
```

can produce only two possible values for x , 101 or 121. The other possibilities are eliminated, because the execution of P_1 and P_2 cannot be interleaved.

Here is a version of the `make_account` function from section 3.1.1, where the deposits and withdrawals have been serialized:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
}
```

```

const protector = make_serializer();
function dispatch(m) {
    return m === "withdraw"
        ? protector(withdraw)
        : m === "deposit"
        ? protector(deposit)
        : m === "balance"
        ? balance
        : Error("Unknown request in make_account",
            m);
}
return dispatch;
}

```

With this implementation, two processes cannot be withdrawing from or depositing into a single account concurrently. This eliminates the source of the error illustrated in figure 3.29, where Peter changes the account balance between the times when Paul accesses the balance to compute the new value and when Paul actually performs the assignment. On the other hand, each account has its own serializer, so that deposits and withdrawals for different accounts can proceed concurrently.

Exercise 3.39

Which of the five possibilities in the parallel execution shown above remain if we instead serialize execution as follows:

```

let x = 10;

const s = make_serializer();

parallel_execute( () => { x = s( () => x * x ); },
    s( () => { x = x + 1; } ) );

```

Exercise 3.40

Give all possible values of x that can result from executing

```

let x = 10;

parallel_execute( () => { x = x * x; },
    () => { x = x * x * x; } );

```

Which of these possibilities remain if we instead use serialized functions:

```
let x = 10;
```

```
const s = make_serializer();

parallel_execute( s( () => x = x * x ) ,
                  s( () => x = x * x * x ) )
```

Exercise 3.41

Ben Bitdiddle worries that it would be better to implement the bank account as follows (where the commented line has been changed):

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    const protected_ = make_serializer();
    function dispatch(m) {
        return m === "withdraw"
            ? protected_(withdraw)
            : m === "deposit"
                ? protected_(deposit)
                : m === "balance"
                    ? protected_( () => balance )() // serialized
                    : error("Unknown request in make_account",
                           m);
    }
}
return dispatch;
```

because allowing unserialized access to the bank balance can result in anomalous behavior. Do you agree? Is there any scenario that demonstrates Ben's concern?

Exercise 3.42

Ben Bitdiddle suggests that it's a waste of time to create a new serialized function in response to every withdraw and deposit message. He says that make_account could be changed so that the calls to protected_ are done outside the dispatch function. That is, an account would

return the same serialized function (which was created at the same time as the account) each time it is asked for a withdrawal function.

```
function make_account(balance) {
  function withdraw(amount) {
    if (balance > amount) {
      balance = balance - amount;
      return balance;
    } else {
      return "Insufficient funds";
    }
  }
  function deposit(amount) {
    balance = balance + amount;
    return balance;
  }
  const protected_ = make_serializer();
  const protected_withdraw = protected_(withdraw);
  const protected_deposit = protected_(deposit);
  function dispatch(m) {
    return m === "withdraw"
      ? protected_withdraw
      : m === "deposit"
        ? protected_deposit
        : m === "balance"
          ? balance
          : error("Unknown request in make_account",
            m);
  }
  return dispatch;
}
```

Is this a safe change to make? In particular, is there any difference in what concurrency is allowed by these two versions of `make_account`?

Complexity of using multiple shared resources

Serializers provide a powerful abstraction that helps isolate the complexities of concurrent programs so that they can be dealt with carefully and (hopefully) correctly. However, while using serializers is relatively straightforward when there is only a single shared resource (such as a single bank account), concurrent programming can be treacherously difficult when there are multiple shared resources.

To illustrate one of the difficulties that can arise, suppose we wish to swap the balances in two bank accounts. We access each account to find the balance, compute the difference between the balances, withdraw this difference from one account, and deposit it in the other account.

We could implement this as follows:⁴⁰

```
function exchange(account1, account2) {
    const difference = account1("balance") - account2("balance");
    account1("withdraw")(difference);
    account2("deposit")(difference);
}
```

This function works well when only a single process is trying to do the exchange. Suppose, however, that Peter and Paul both have access to accounts a_1 , a_2 , and a_3 , and that Peter exchanges a_1 and a_2 while Paul concurrently exchanges a_1 and a_3 . Even with account deposits and withdrawals serialized for individual accounts (as in the `make_account` function shown above in this section), `exchange` can still produce incorrect results. For example, Peter might compute the difference in the balances for a_1 and a_2 , but then Paul might change the balance in a_1 before Peter is able to complete the exchange.⁴¹ For correct behavior, we must arrange for the `exchange` function to lock out any other concurrent accesses to the accounts during the entire time of the exchange.

One way we can accomplish this is by using both accounts' serializers to serialize the entire `exchange` function. To do this, we will arrange for access to an account's serializer. Note that we are deliberately breaking the modularity of the bank-account object by exposing the serializer. The following version of `make_account` is identical to the original version given in Section 3.1.1, except that a serializer is provided to protect the `balance` variable, and the serializer is exported via message passing:

```
function make_account_and_serializer(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
        } else {
            "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    const balance_serializer = make_serializer();
    function dispatch(m) {
        if (m === "withdraw") {
            return withdraw;
        } else if (m === "deposit") {
```

⁴⁰We have simplified `exchange` by exploiting the fact that our `deposit` message accepts negative amounts. (This is a serious bug in our banking system!)

⁴¹If the account balances start out as \$10, \$20, and \$30, then after any number of concurrent exchanges, the balances should still be \$10, \$20, and \$30 in some order. Serializing the deposits to individual accounts is not sufficient to guarantee this. See exercise 3.43.

```

        return deposit;
    } else if (m === "balance") {
        return balance;
    } else if (m === "serializer") {
        return balance_serializer;
    } else {
        return "Unknown request -- MAKE-ACCOUNT";
    }
}
return dispatch;
}

```

We can use this to do serialized deposits and withdrawals. However, unlike our earlier serialized account, it is now the responsibility of each user of bank-account objects to explicitly manage the serialization, for example as follows:⁴²

```

function deposit(account, amount) {
    const s = account("serializer");
    const d = account("deposit");
    s(d(amount));
}

```

Exporting the serializer in this way gives us enough flexibility to implement a serialized exchange program. We simply serialize the original exchange function with the serializers for both accounts:

```

function serialized_exchange(account1, account2) {
    const serializer1 = account1("serializer");
    const serializer2 = account2("serializer");
    serializer1(serializer2(exchange))(account1, account2);
}

```

Exercise 3.43

Suppose that the balances in three accounts start out as \$10, \$20, and \$30, and that multiple processes run, exchanging the balances in the accounts. Argue that if the processes are run sequentially, after any number of concurrent exchanges, the account balances should be \$10, \$20, and \$30 in some order. Draw a timing diagram like the one in Figure 3.29 to show how this condition can be violated if the exchanges are implemented using the first version of the account-exchange program in this section. On the other hand, argue that even with this exchange program, the sum of the balances in the accounts will be preserved. Draw a timing diagram to show how even this condition would be violated if we did not serialize the transactions on individual accounts.

⁴²Exercise 3.45 investigates why deposits and withdrawals are no longer automatically serialized by the account.

Exercise 3.44

Consider the problem of transferring an amount from one account to another. Ben Bitdiddle claims that this can be accomplished with the following function, even if there are multiple people concurrently transferring money among multiple accounts, using any account mechanism that serializes deposit and withdrawal transactions, for example, the version of make_account in the text above.

```
function transfer(from_account, to_account, amount) {
  from_account("withdraw")(amount);
  to_account("deposit")(amount);
}
```

Louis Reasoner claims that there is a problem here, and that we need to use a more sophisticated method, such as the one required for dealing with the exchange problem. Is Louis right? If not, what is the essential difference between the transfer problem and the exchange problem? (You should assume that the balance in from_account is at least amount.)

Exercise 3.45

Louis Reasoner thinks our bank-account system is unnecessarily complex and error-prone now that deposits and withdrawals aren't automatically serialized. He suggests that make_account_and_serializer should have exported the serializer (for use by such functions as serialized_exchange) in addition to (rather than instead of) using it to serialize accounts and deposits as make_account did. He proposes to redefine accounts as follows:

```
function make_account_and_serializer(balance) {
  function withdraw(amount) {
    if (balance > amount) {
      balance = balance - amount;
    } else {
      "Insufficient funds";
    }
  }
  function deposit(amount) {
    balance = balance + amount;
    return balance;
  }
  const balance_serializer = make_serializer();
  function dispatch(m) {
    return m === "withdraw"
      ? balance_serializer(withdraw)
      : m === "deposit"
        ? balance_serializer(deposit)
        : m === "balance"
          ? balance
          : undefined;
  }
}
```

```

        : m === "serializer"
        ? balance_serializer
        : error("Unknown request in make_account",
            m);
    }
    return dispatch;
}

```

Then deposits are handled as with the original `make_account`:

```

function deposit(account, amount) {
    account("deposit")(amount);
}

```

Explain what is wrong with Louis's reasoning. In particular, consider what happens when `serialized_exchange` is called.

Implementing serializers

We implement serializers in terms of a more primitive synchronization mechanism called a *mutex*. A mutex is an object that supports two operations—the mutex can be *acquired*, and the mutex can be *released*. Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released.⁴³ In our implementation, each serializer has an associated mutex. Given a function `p`, the serializer returns a function that acquires the mutex, runs `p`, and then releases the mutex. This ensures that only one of the functions produced by the serializer can be running at once, which is precisely the serialization property that we need to guarantee.

```

function make_serializer() {
    const mutex = make_mutex();
    return p => {
        function serialized_p(args) {
            mutex("acquire");
            const val = p(args);
            mutex("release");
            return val;
        }
        return serialized_p;
    };
}

```

⁴³The term ‘mutex’ is an abbreviation for *mutual exclusion*. The general problem of arranging a mechanism that permits concurrent processes to safely share resources is called the mutual exclusion problem. Our mutex is a simple variant of the *semaphore* mechanism (see exercise 3.47), which was introduced in the ‘THE’ Multi-programming System developed at the Technological University of Eindhoven and named for the university’s initials in Dutch (Dijkstra 1968a). The acquire and release operations were originally called P and V, from the Dutch words *passeren* (to pass) and *vrijgeven* (to release), in reference to the semaphores used on railroad systems. Dijkstra’s classic exposition (1968b) was one of the first to clearly present the issues of concurrency control, and showed how to use semaphores to handle a variety of concurrency problems.

The mutex is a mutable object (here we'll use a one-element list, which we'll refer to as a *cell*) that can hold the value true or false. When the value is false, the mutex is available to be acquired. When the value is true, the mutex is unavailable, and any process that attempts to acquire the mutex must wait.

Our mutex constructor `make_mutex` begins by initializing the cell contents to false. To acquire the mutex, we test the cell. If the mutex is available, we set the cell contents to true and proceed. Otherwise, we wait in a loop, attempting to acquire over and over again, until we find that the mutex is available.⁴⁴ To release the mutex, we set the cell contents to false.

```
function make_mutex() {
    const cell = list(false);
    function the_mutex(m) {
        return m === "aquire"
            ? ( test_and_set(cell)
                ? the_mutex("aquire") // retry
                : true )
            : m === "release"
            ? clear(cell)
            : error("Unknown request in mutex",
                    m);
    }
    return the_mutex;
}
function clear(cell) {
    set_head(cell, false);
}
```

The function `test_and_set` tests the cell and returns the result of the test. In addition, if the test was false, `test_and_set` sets the cell contents to true before returning false. We can express this behavior as the following function:

```
function test_and_set(cell) {
    if (head(cell)) {
        return true;
    } else {
        set_head(cell, true);
        return false;
    }
}
```

However, this implementation of `test_and_set` does not suffice as it stands. There is a crucial subtlety here, which is the essential place where concurrency control enters the system: The `test_and_set` operation must be performed *atomically*. That is, we must guarantee that, once

⁴⁴In most time-shared operating systems, processes that are blocked by a mutex do not waste time ‘busy-waiting’ as above. Instead, the system schedules another process to run while the first is waiting, and the blocked process is awakened when the mutex becomes available.

a process has tested the cell and found it to be false, the cell contents will actually be set to true before any other process can test the cell. If we do not make this guarantee, then the mutex can fail in a way similar to the bank-account failure in Figure 3.29. (See exercise 3.46.)

The actual implementation of `test_and_set` depends on the details of how our system runs concurrent processes. For example, we might be executing concurrent processes on a sequential processor using a time-slicing mechanism that cycles through the processes, permitting each process to run for a short time before interrupting it and moving on to the next process. In that case, `test_and_set` can work by disabling time slicing during the testing and setting. Alternatively, multiprocessing computers provide instructions that support atomic operations directly in hardware.⁴⁵

Exercise 3.46

Suppose that we implement `test_and_set` using an ordinary function as shown in the text, without attempting to make the operation atomic. Draw a timing diagram like the one in figure 3.29 to demonstrate how the mutex implementation can fail by allowing two processes to acquire the mutex at the same time.

Exercise 3.47

A semaphore (of size n) is a generalization of a mutex. Like a mutex, a semaphore supports acquire and release operations, but it is more general in that up to n processes can acquire it concurrently. Additional processes that attempt to acquire the semaphore must wait for release operations. Give implementations of semaphores

- a. in terms of mutexes
- b. in terms of atomic `test_and_set` operations.

⁴⁵There are many variants of such instructions—including `test-and-set`, `test-and-clear`, `swap`, `compare-and-exchange`, `load-reserve`, and `store-conditional`—whose design must be carefully matched to the machine’s processor–memory interface. One issue that arises here is to determine what happens if two processes attempt to acquire the same resource at exactly the same time by using such an instruction. This requires some mechanism for making a decision about which process gets control. Such a mechanism is called an *arbiter*. Arbiters usually boil down to some sort of hardware device. Unfortunately, it is possible to prove that one cannot physically construct a fair arbiter that works 100% of the time unless one allows the arbiter an arbitrarily long time to make its decision. The fundamental phenomenon here was originally observed by the fourteenth-century French philosopher Jean Buridan in his commentary on Aristotle’s *De caelo*. Buridan argued that a perfectly rational dog placed between two equally attractive sources of food will starve to death, because it is incapable of deciding which to go to first.

Deadlock

Now that we have seen how to implement serializers, we can see that account exchanging still has a problem, even with the `serialized_exchange` function above. Imagine that Peter attempts to exchange a_1 with a_2 while Paul concurrently attempts to exchange a_2 with a_1 . Suppose that Peter's process reaches the point where it has entered a serialized function protecting a_1 and, just after that, Paul's process enters a serialized function protecting a_2 . Now Peter cannot proceed (to enter a serialized function protecting a_2) until Paul exits the serialized function protecting a_2 . Similarly, Paul cannot proceed until Peter exits the serialized function protecting a_1 . Each process is stalled forever, waiting for the other. This situation is called a *deadlock*. Deadlock is always a danger in systems that provide concurrent access to multiple shared resources.

One way to avoid the deadlock in this situation is to give each account a unique identification number and rewrite `serialized_exchange` so that a process will always attempt to enter a function protecting the lowest-numbered account first. Although this method works well for the exchange problem, there are other situations that require more sophisticated deadlock-avoidance techniques, or where deadlock cannot be avoided at all. (See exercises 3.48 and 3.49.)⁴⁶

Exercise 3.48

Explain in detail why the deadlock-avoidance method described above, (i.e., the accounts are numbered, and each process attempts to acquire the smaller-numbered account first) avoids deadlock in the exchange problem. Rewrite `serialized_exchange` to incorporate this idea. (You will also need to modify `make_account` so that each account is created with a number, which can be accessed by sending an appropriate message.)

Exercise 3.49

Give a scenario where the deadlock-avoidance mechanism described above does not work. (Hint: In the exchange problem, each process knows in advance which accounts it will need to get access to. Consider a situation where a process must get access to some shared resources before it can know which additional shared resources it will require.)

⁴⁶The general technique for avoiding deadlock by numbering the shared resources and acquiring them in order is due to Havender (1968). Situations where deadlock cannot be avoided require *deadlock-recovery* methods, which entail having processes ‘back out’ of the deadlocked state and try again. Deadlock-recovery mechanisms are widely used in database management systems, a topic that is treated in detail in Gray and Reuter 1993.

Concurrency, time, and communication

We've seen how programming concurrent systems requires controlling the ordering of events when different processes access shared state, and we've seen how to achieve this control through judicious use of serializers. But the problems of concurrency lie deeper than this, because, from a fundamental point of view, it's not always clear what is meant by 'shared state.'

Mechanisms such as `test_and_set` require processes to examine a global shared flag at arbitrary times. This is problematic and inefficient to implement in modern high-speed processors, where due to optimization techniques such as pipelining and cached memory, the contents of memory may not be in a consistent state at every instant. In contemporary multiprocessing systems, therefore, the serializer paradigm is being supplanted by new approaches to concurrency control.⁴⁷

The problematic aspects of shared state also arise in large, distributed systems. For instance, imagine a distributed banking system where individual branch banks maintain local values for bank balances and periodically compare these with values maintained by other branches. In such a system the value of 'the account balance' would be undetermined, except right after synchronization. If Peter deposits money in an account he holds jointly with Paul, when should we say that the account balance has changed—when the balance in the local branch changes, or not until after the synchronization? And if Paul accesses the account from a different branch, what are the reasonable constraints to place on the banking system such that the behavior is 'correct'? The only thing that might matter for correctness is the behavior observed by Peter and Paul individually and the 'state' of the account immediately after synchronization. Questions about the 'real' account balance or the order of events between synchronizations may be irrelevant or meaningless.⁴⁸

The basic phenomenon here is that synchronizing different processes, establishing shared state, or imposing an order on events requires communication among the processes. In essence, any notion of time in concurrency control must be intimately tied to communication.⁴⁹ It is intriguing that a similar connection between time and communication also arises in the Theory of Relativity, where the speed of light (the fastest signal that can be used to synchronize

⁴⁷One such alternative to serialization is called *barrier synchronization*. The programmer permits concurrent processes to execute as they please, but establishes certain synchronization points ('barriers') through which no process can proceed until all the processes have reached the barrier. Modern processors provide machine instructions that permit programmers to establish synchronization points at places where consistency is required. The PowerPC™, for example, includes for this purpose two instructions called SYNC and EIEIO (Enforced In-order Execution of Input/Output).

⁴⁸This may seem like a strange point of view, but there are systems that work this way. International charges to credit-card accounts, for example, are normally cleared on a per-country basis, and the charges made in different countries are periodically reconciled. Thus the account balance may be different in different countries.

⁴⁹For distributed systems, this perspective was pursued by Lamport (1978), who showed how to use communication to establish 'global clocks' that can be used to establish orderings on events in distributed systems.

events) is a fundamental constant relating time and space. The complexities we encounter in dealing with time and state in our computational models may in fact mirror a fundamental complexity of the physical universe.

3.5 Streams

We've gained a good understanding of assignment as a tool in modeling, as well as an appreciation of the complex problems that assignment raises. It is time to ask whether we could have gone about things in a different way, so as to avoid some of these problems. In this section, we explore an alternative approach to modeling state, based on data structures called *streams*. As we shall see, streams can mitigate some of the complexity of modeling state.

Let's step back and review where this complexity comes from. In an attempt to model real-world phenomena, we made some apparently reasonable decisions: We modeled real-world objects with local state by computational objects with local variables. We identified time variation in the real world with time variation in the computer. We implemented the time variation of the states of the model objects in the computer with assignments to the local variables of the model objects.

Is there another approach? Can we avoid identifying time in the computer with time in the modeled world? Must we make the model change with time in order to model phenomena in a changing world? Think about the issue in terms of mathematical functions. We can describe the time-varying behavior of a quantity x as a function of time $x(t)$. If we concentrate on x instant by instant, we think of it as a changing quantity. Yet if we concentrate on the entire time history of values, we do not emphasize change—the function itself does not change.⁵⁰

If time is measured in discrete steps, then we can model a time function as a (possibly infinite) sequence. In this section, we will see how to model change in terms of sequences that represent the time histories of the systems being modeled. To accomplish this, we introduce new data structures called *streams*. From an abstract point of view, a stream is simply a sequence. However, we will find that the straightforward implementation of streams as lists (as in section 2.2.1) doesn't fully reveal the power of stream processing. As an alternative, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) sequences as streams.

Stream processing lets us model systems that have state without ever using assignment or mutable data. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment. On the other

⁵⁰Physicists sometimes adopt this view by introducing the ‘world lines’ of particles as a device for reasoning about motion. We've also already mentioned (section 2.2.3) that this is the natural way to think about signal-processing systems. We will explore applications of streams to signal processing in section 3.5.3.

hand, the stream framework raises difficulties of its own, and the question of which modeling technique leads to more modular and more easily maintained systems remains open.

3.5.1 Streams Are Delayed Lists

As we saw in section 2.2.3, sequences can serve as standard interfaces for combining program modules. We formulated powerful abstractions for manipulating sequences, such as `map`, `filter`, and `accumulate`, that capture a wide variety of operations in a manner that is both succinct and elegant.

Unfortunately, if we represent sequences as lists, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations. When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

To see why this is true, let us compare two programs for computing the sum of all the prime numbers in an interval. The first program is written in standard iterative style:⁵¹

```
function sum_primes(a, b) {
    function iter(count, accum) {
        if (count > b) {
            return accum;
        } else {
            if (is_prime(count)) {
                return iter(count + 1, count + accum);
            } else {
                return iter(count + 1, accum);
            }
        }
    }
    return iter(a, 0);
}
```

The second program performs the same computation using the sequence operations of section 2.2.3:

```
function sum_primes(a, b) {
    return accumulate((x, y) => x + y,
                      0,
                      filter(is_prime,
                             enumerate_interval(a, b)));
}
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, the filter in the second program cannot do any testing until `enumerate_interval`

⁵¹Assume that we have a predicate `is_prime` (e.g., as in section 1.2.6) that tests for primality.

has constructed a complete list of the numbers in the interval. The filter generates another list, which in turn is passed to accumulate before being collapsed to form a sum. Such large intermediate storage is not needed by the first program, which we can think of as enumerating the interval incrementally, adding each prime to the sum as it is generated.

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

```
head(tail(filter(is_prime,
    enumerate_interval(10000, 1000000))));
```

This expression does find the second prime when given enough time and space, but the computational overhead is outrageous. We construct a list of almost a million integers, filter this list by testing each element for primality, and then ignore almost all of the result. In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.

Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists. With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation. The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists. In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

In their most basic form, streams are similar to lists. The empty stream is `null`, a non-empty stream is a pair, and the head of the pair is a data item. However, the tail of a pair that represents a non-empty stream is not a stream, but a *nullary function that returns a stream*. The stream returned by the function, we call *the tail of the stream*. If we have a data item `x` and a stream `s`, we can construct a stream whose head is `x` and whose tail is `s` by evaluating `pair(x, () => s)`.

In order to access the data item of a non-empty stream, we just use `head` as with lists. In order to access the tail of a stream `s`, we need to *apply* `tail(s)`, i.e. evaluate `(tail(s))()`. For convenience, we therefore define

```
function stream_tail(stream) {
    return tail(stream)();
}
```

We can make and use streams, in just the same way as we can make and use lists, to represent aggregate data arranged in a sequence. In particular, we can build stream analogs of the list

operations from chapter 2, such as `list_ref`, `map`, and `for_each`:⁵²

```
function stream_ref(s, n) {
  return n === 0
    ? head(s)
    : stream_ref(stream_tail(s), n - 1);
}
function stream_map(f, s) {
  return is_null(s)
    ? null
    : pair(f(head(s)),
           () => stream_map(f, stream_tail(s)));
}
function stream_for_each(fun, s) {
  if (is_null(s)) {
    return true;
  } else {
    fun(head(s));
    return stream_for_each(fun, stream_tail(s));
  }
}
```

The function `stream_for_each` is useful for viewing streams:

```
function display_stream(s) {
  return stream_for_each(display, s);
}
```

The function that represents the tail of a stream is evaluated when it is accessed, using `stream_tail`. This design choice is reminiscent of our discussion of rational numbers in section 2.1.2, where we saw that we can choose to implement rational numbers so that the reduction of numerator and denominator to lowest terms is performed either at construction time or at selection time. The two rational-number implementations produce the same data abstraction, but the choice has an effect on efficiency. There is a similar relationship between streams and ordinary lists. As a data abstraction, streams are the same as lists. The difference is the time at which the elements are evaluated. With ordinary lists, both the head tail are evaluated at construction time. With streams, the tail is evaluated at selection time.

The tail of a stream is ‘wrapped’ in a function. It is a *delayed expression*, a ‘promise’ to evaluate an expression `exp` at some future time. Correspondingly, `stream_tail` forces the tail to fulfill its promise. It selects the tail of the pair and evaluates the delayed expression found there to obtain the rest of the stream.

⁵²This should bother you. The fact that we are defining such similar functions for streams and lists indicates that we are missing some underlying abstraction. Unfortunately, in order to exploit this abstraction, we will need to exert finer control over the process of evaluation than we can at present. We will discuss this point further at the end of section 3.5.4. In section 4.2, we’ll develop a framework that unifies lists and streams.

Streams in action

To see how this data structure behaves, let us analyze the ‘outrageous’ prime computation we saw above, reformulated in terms of streams:

```
head(stream_tail(stream_filter(
    is_prime,
    stream_enumerate_interval(10000,
        1000000))));
```

We will see that it does indeed work efficiently.

We begin by calling `stream_enumerate_interval` with the arguments 10,000 and 1,000,000. The function `stream_enumerate_interval` is the stream analog of `enumerate_interval` (section 2.2.3):

```
function stream_enumerate_interval(low, high) {
    return low > high
        ? null
        : pair(low,
            () => stream_enumerate_interval(low + 1,
                high));
}
```

and thus the result returned by `stream_enumerate_interval`, formed by the `pair`, is⁵³

```
pair(10000, () => stream_enumerate_interval(10001, 1000000));
```

That is, `stream_enumerate_interval` returns a stream represented as a pair whose head is 10,000 and whose tail is a promise to enumerate more of the interval if so requested. This stream is now filtered for primes, using the stream analog of the `filter` function (section 2.2.3):

```
function stream_filter(pred, s) {
    return is_null(s)
        ? null
        : pred(head(s))
            ? pair(head(s),
                () => stream_filter(pred,
                    stream_tail(s)))
            : stream_filter(pred,
                stream_tail(s));
}
```

The function `stream_filter` tests the head of the stream (which is 10,000). Since this is not prime, `stream_filter` examines the tail of its input stream. The call to `stream_tail` forces evaluation of the delayed `stream_enumerate_interval`, which now returns

⁵³The numbers shown here do not really appear in the delayed expression. What actually appears is the original expression, in an environment in which the variables are bound to the appropriate numbers. For example, `low + 1` with `low` bound to 10,000 actually appears where 10001 is shown.

```
pair(10001, () => stream_enumerate_interval(10002, 1000000));
```

The function `stream_filter` now looks at the head of this stream, 10,001, sees that this is not prime either, forces another `stream_tail`, and so on, until `stream_enumerate_interval` yields the prime 10,007, whereupon `stream_filter`, according to its definition, returns

```
pair(head(stream), stream_filter(pred, stream_tail(stream)));
```

which in this case is

```
pair(10007,
    () => stream_filter(is_prime,
        pair(10008,
            () => stream_enumerate_interval(10009,
                1000000)))
    )
);
```

This result is now passed to `stream_tail` in our original expression. This forces the delayed `stream_filter`, which in turn keeps forcing the delayed `stream_enumerate_interval` until it finds the next prime, which is 10,009. Finally, the result passed to `head` in our original expression is

```
pair(10009,
    () => stream_filter(is_prime,
        pair(10010,
            () => stream_enumerate_interval(10011,
                1000000)))
    )
);
```

The function `head` returns 10,009, and the computation is complete. Only as many integers were tested for primality as were necessary to find the second prime, and the interval was enumerated only as far as was necessary to feed the prime filter.

In general, we can think of delayed evaluation as ‘demand-driven’ programming, whereby each stage in the stream process is activated only enough to satisfy the next stage. What we have done is to decouple the actual order of events in the computation from the apparent structure of our functions. We write functions as if the streams existed ‘all at once’ when, in reality, the computation is performed incrementally, as in traditional programming styles.

An optimization

When we construct stream pairs, we delay the evaluation of their tail expressions by wrapping these expressions in a function. We force their evaluation when needed, by applying the function.

This implementation suffices for streams to work as advertised, but there is an important optimization that we can include. In many applications, we end up forcing the same delayed object many times. This can lead to serious inefficiency in recursive programs involving streams. (See exercise 3.57.) The solution is to build delayed objects so that the first time they are forced, they store the value that is computed. Subsequent forcings will simply return the stored value without repeating the computation. In other words, we implement the construction of stream pairs as a memoized function similar to the one described in exercise 3.27. One way to accomplish this is to use the following function, which takes as argument a function (of no arguments) and returns a memoized version of the function. The first time the memoized function is run, it saves the computed result. On subsequent evaluations, it simply returns the result.

```
function memo(fun) {
  let already_run = false;
  let result = undefined;
  return () => {
    if (!already_run) {
      result = fun();
      already_run = true;
      return result;
    } else {
      return result;
    }
  };
}
```

We can make use of `memo` whenever we construct a stream pair. For example, instead of

```
function stream_map(f, s) {
  return is_null(s)
    ? null
    : pair(f(head(s)),
           () => stream_map(f, stream_tail(s)));
}
```

we can define an optimized function `stream_map` as follows:⁵⁴

⁵⁴There are many possible implementations of streams other than the one described in this section. Delayed evaluation, which is the key to making streams practical, was inherent in Algol 60's *call-by-name* parameter-passing method. The use of this mechanism to implement streams was first described by Landin (1965). Delayed evaluation for streams was introduced into Lisp by Friedman and Wise (1976). In their implementation, `cons` always delays evaluating its arguments, so that lists automatically behave as streams. The memoizing optimization is also known as *call-by-need*. The Algol community would refer to our original delayed objects as *call-by-name*.

```
function stream_map_optimized(f, s) {
  return is_null(s)
    ? null
    : pair(f(head(s)),
            memo( () => stream_map_optimized(
                  f, stream_tail(s)) ));
}
```

Exercise 3.50

Define a function *stream_combine* that takes a binary function and two streams as arguments and returns a stream whose elements are the results of applying the function pairwise to the corresponding elements of the argument streams.

```
function stream_combine(f, s1, s2) {
  ...
}
```

Exercise 3.51

In order to take a closer look at delayed evaluation, we will use the following function, which simply returns its argument after printing it:

```
function show(x) {
  display(x);
  return x;
}
```

What does the interpreter print in response to evaluating each expression in the following sequence?⁵⁵

```
let x = stream_map(show, stream_enumerate_interval(0, 10));
stream_ref(x, 5);
stream_ref(x, 7);
```

What does the evaluator print if *stream_map_optimized* is used instead of *stream_map*?

thunks and to the optimized versions as *call-by-need thunks*.

⁵⁵Exercises such as 3.51 and 3.52 are valuable for testing our understanding of how delayed evaluation works. On the other hand, intermixing delayed evaluation with printing—and, even worse, with assignment—is extremely confusing, and instructors of courses on computer languages have traditionally tormented their students with examination questions such as the ones in this section. Needless to say, writing programs that depend on such subtleties is odious programming style. Part of the power of stream processing is that it lets us ignore the order in which events actually happen in our programs. Unfortunately, this is precisely what we cannot afford to do in the presence of assignment, which forces us to be concerned with time and change.

```
let x = stream_map_optimized(
    show,
    stream_enumerate_interval(0, 10));
stream_ref(x, 5);
stream_ref(x, 7);
```

Exercise 3.52

Consider the sequence of expressions

```
let sum = 0;

function accum(x) {
    sum = x + sum;
    return sum;
}

const seq = stream_map(
    accum,
    stream_enumerate_interval(1, 20));
const y = stream_filter(is_even, seq);

const z = stream_filter(x => x % 5 === 0, seq);

stream_ref(y, 7);

display_stream(z);
```

What is the value of `sum` after each of the above expressions is evaluated? What is the printed response to evaluating the `stream_ref` and `display_stream` expressions? Would these responses differ if we had applied the function `memo` on every tail of every constructed stream pair, as suggested in the optimization above? Explain.

3.5.2 Infinite Streams

We have seen how to support the illusion of manipulating streams as complete entities even though, in actuality, we compute only as much of the stream as we need to access. We can exploit this technique to represent sequences efficiently as streams, even if the sequences are very long. What is more striking, we can use streams to represent sequences that are infinitely long. For instance, consider the following definition of the stream of positive integers:

```
function integers_starting_from(n) {
    return pair(n,
        () => integers_starting_from(n + 1)
    );
}
```

This makes sense because `integers` will be a pair whose head is 1 and whose tail is a promise to produce the integers beginning with 2. This is an infinitely long stream, but in any given time we can examine only a finite portion of it. Thus, our programs will never know that the entire infinite stream is not there.

Using `integers` we can define other infinite streams, such as the stream of integers that are not divisible by 7:

```
function is_divisible(x, y) {
    return x % y === 0;
}

const no_sevens =
    stream_filter(x => ! is_divisible(x, 7),
        integers);
```

Then we can find integers not divisible by 7 simply by accessing elements of this stream:

```
stream_ref(no_sevens, 100);
```

In analogy with `integers`, we can define the infinite stream of Fibonacci numbers:

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}

const fibs = fibgen(0, 1);
```

The function `fibs` is a pair whose head is 0 and whose tail is a promise to evaluate `fibgen(1, 1)`. When we evaluate this delayed `fibgen(1, 1)`, it will produce a pair whose head is 1 and whose tail is a promise to evaluate `fibgen(1, 2)`, and so on.

For a look at a more exciting infinite stream, we can generalize the `no_sevens` example to construct the infinite stream of prime numbers, using a method known as the *sieve of Eratosthenes*.⁵⁶

⁵⁶Eratosthenes, a third-century B.C. Alexandrian Greek philosopher, is famous for giving the first accurate

We start with the integers beginning with 2, which is the first prime. To get the rest of the primes, we start by filtering the multiples of 2 from the rest of the integers. This leaves a stream beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of this stream. This leaves a stream beginning with 5, which is the next prime, and so on. In other words, we construct the primes by a sieving process, described as follows: To sieve a stream S , form a stream whose first element is the first element of S and the rest of which is obtained by filtering all multiples of the first element of S out of the rest of S and sieving the result. This process is readily described in terms of stream operations:

```
function sieve(stream) {
    return pair(head(stream),
        () => sieve(stream_filter(
            x => !is_divisible(x,
                head(stream)),
            stream_tail(stream)
        )
    )
);
}
```

Now to find a particular prime we need only ask for it:

```
stream_ref(primes, 50);
```

It is interesting to contemplate the signal-processing system set up by `sieve`, shown in the ‘Henderson diagram’ in Figure 3.31.⁵⁷ The input stream feeds into an ‘unpairer’ that separates the first element of the stream from the rest of the stream. The first element is used to construct a divisibility filter, through which the rest is passed, and the output of the filter is fed to another sieve box. Then the original first element is paired onto the output of the internal sieve to form the output stream. Thus, not only is the stream infinite, but the signal processor is also infinite, because the sieve contains a sieve within it.

estimate of the circumference of the Earth, which he computed by observing shadows cast at noon on the day of the summer solstice. Eratosthenes’s sieve method, although ancient, has formed the basis for special-purpose hardware ‘sieves’ that, until the 1970s, were the most powerful tools in existence for locating large primes. Since then, however, these methods have been superseded by outgrowths of the probabilistic techniques discussed in section 1.2.6.

⁵⁷We have named these figures after Peter Henderson, who was the first person to show us diagrams of this sort as a way of thinking about stream processing. Each solid line represents a stream of values being transmitted. The dashed line from the head to the pair and the filter indicates that this is a single value rather than a stream.

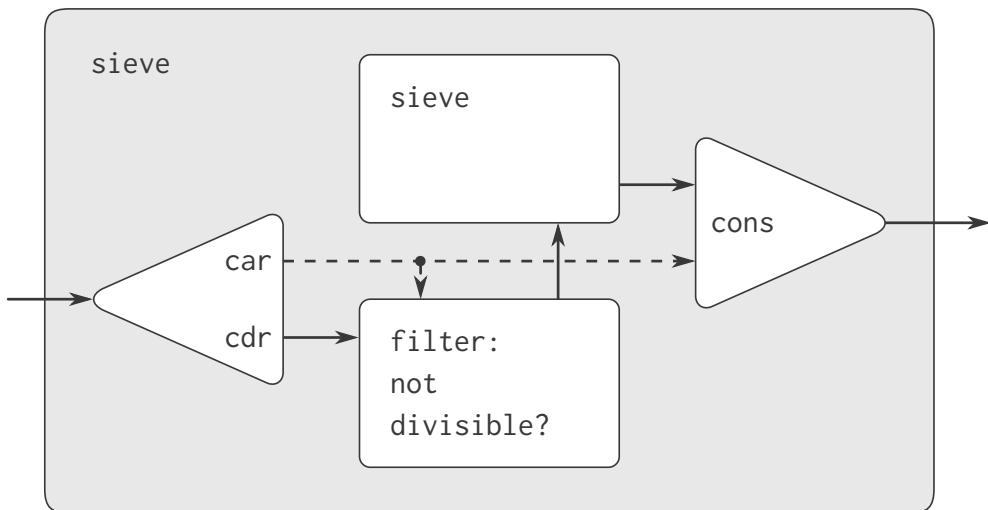


Figure 3.31: The prime sieve viewed as a signal-processing system.

Defining streams implicitly

The integers and fibs streams above were defined by specifying ‘generating’ functions that explicitly compute the stream elements one by one. An alternative way to specify streams is to take advantage of delayed evaluation to define streams implicitly. For example, the following expression defines the stream ones to be an infinite stream of ones:

```
const ones = pair(1, () => ones);
```

This works much like the definition of a recursive function: ones is a pair whose head is 1 and whose tail is a promise to evaluate ones. Evaluating the tail gives us again a 1 and a promise to evaluate ones, and so on.

We can do more interesting things by manipulating streams with operations such as `add_streams`, which produces the elementwise sum of two given streams.⁵⁸

```
function add_streams(s1, s2) {
  return stream_combine((x1, x2) => x1 + x2, s1, s2);
}
```

Now we can define the integers as follows:

```
const integers = pair(1, () => add_streams(ones, integers));
```

This defines integers to be a stream whose first element is 1 and the rest of which is the sum of ones and integers. Thus, the second element of integers is 1 plus the first element of integers, or 2; the third element of integers is 1 plus the second element of integers, or 3; and so on. This definition works because, at any point, enough of the integers stream has been generated so that we can feed it back into the definition to produce the next integer.

⁵⁸This uses the function `stream_merge` from exercise 3.50.

We can define the Fibonacci numbers in the same style:

```
const fibs = pair(0,
    () => pair(1,
        () => add_streams(stream_tail(
            fibs))
    )
);
```

This definition says that `fibs` is a stream beginning with 0 and 1, such that the rest of the stream can be generated by adding `fibs` to itself shifted by one place:

$$\begin{array}{cccccccccc} 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & \dots & = \text{stream_tail}(\text{fibs}) \\ 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & \dots & = \text{fibs} \\ \hline 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & \dots = \text{fibs} \end{array}$$

The function `scale_stream` is also useful in formulating such stream definitions. This multiplies each item in a stream by a given constant:

```
function scale_stream(stream, factor) {
    return stream_map(x => x * factor,
        stream);
}
```

For example,

```
const double = pair(1, () => scale_stream(double, 2));
```

produces the stream of powers of 2: 1, 2, 4, 8, 16, 32, ...

An alternate definition of the stream of primes can be given by starting with the integers and filtering them by testing for primality. We will need the first prime, 2, to get started:

```
const primes = pair(2,
    () => stream_filter(
        is_prime,
        integers_starting_from(3))
);
```

This definition is not so straightforward as it appears, because we will test whether a number n is prime by checking whether n is divisible by a prime (not by just any integer) less than or equal to \sqrt{n} :

```
function is_prime(n) {
    function iter(ps) {
        return square(head(ps)) > n
            ? true
            : is_divisible(n, head(ps))
            ? false
            : iter(stream_tail(ps));
    }
}
```

```

    return iter(primes);
}

```

This is a recursive definition, since `primes` is defined in terms of the `is_prime` predicate, which itself uses the `primes` stream. The reason this function works is that, at any point, enough of the `primes` stream has been generated to test the primality of the numbers we need to check next. That is, for every n we test for primality, either n is not prime (in which case there is a prime already generated that divides it) or n is prime (in which case there is a prime already generated—i.e., a prime less than n —that is greater than \sqrt{n}).⁵⁹

Exercise 3.53

Without running the program, describe the elements of the stream defined by

```
const s = pair(1, () => add_streams(s, s));
```

Exercise 3.54

Define a function `mul_streams`, analogous to `add_streams`, that produces the elementwise product of its two input streams. Use this together with the stream of integers to complete the following definition of the stream whose n th element (counting from 0) is $n + 1$ factorial:

```

// mul_streams to be written by students
const factorials = pair(1, () => mul_streams(???, ???));

```

Exercise 3.55

Define a function `partial_sums` that takes as argument a stream S and returns the stream whose elements are $S_0, S_0 + S_1, S_0 + S_1 + S_2, \dots$. For example, `partial_sums(integers)` should be the stream $1, 3, 6, 10, 15, \dots$.

Exercise 3.56

A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them

⁵⁹This last point is very subtle and relies on the fact that $p_{n+1} \leq p_n^2$. (Here, p_k denotes the k th prime.) Estimates such as these are very difficult to establish. The ancient proof by Euclid that there are an infinite number of primes shows that $p_{n+1} \leq p_1 p_2 \cdots p_n + 1$, and no substantially better result was proved until 1851, when the Russian mathematician P. L. Chebyshev established that $p_{n+1} \leq 2p_n$ for all n . This result, originally conjectured in 1845, is known as *Bertrand's hypothesis*. A proof can be found in section 22.3 of Hardy and Wright 1960.

fit the requirement. As an alternative, let us call the required stream of numbers S and notice the following facts about it.

- S begins with 1.
- The elements of $\text{scale_stream}(S, 2)$ are also elements of S .
- The same is true for $\text{scale_stream}(S, 3)$ and $\text{scale_stream}(5, S)$.
- These are all the elements of S .

Now all we have to do is combine elements from these sources. For this we define a function merge that combines two ordered streams into one ordered result stream, eliminating repetitions:

```
function merge(s1, s2) {
  if (is_null(s1)) {
    return s2;
  } else if (is_null(s2)) {
    return s1;
  } else {
    const s1head = head(s1);
    const s2head = head(s2);
    if (s1head < s2head) {
      return pair(s1head,
                  () => merge(stream_tail(s1), s2)
                );
    } else if (s1head > s2head) {
      return pair(s2head,
                  () => merge(s1, stream_tail(s2))
                );
    } else {
      return merge(stream_tail(s1), stream_tail(s2));
    }
  }
}
```

Then the required stream may be constructed with merge , as follows:

```
const S = pair(1, () => merge(??, ??));
```

Fill in the missing expressions in the places marked ?? above.

Exercise 3.57

How many additions are performed when we compute the n th Fibonacci number using the definition of fibs based on the add_streams function, implemented using $\text{pair}(\dots, () => \dots)$

as described in the beginning of section 3.5.1? Show that the number of additions is exponentially greater than if we had implemented `add_streams` using the optimization using `pair(..., memo(() => ...))` described in the last part of section 3.5.1.⁶⁰

Exercise 3.58

Give an interpretation of the stream computed by the function :

```
function expand(num, den, radix) {
    return pair(quotient(num * radix, den),
               expand((num * radix) % den, den, radix));
}
```

where the function `quotient` computes integer division, in which the fractional part (remainder) is discarded. What are the successive elements produced by `expand(1, 7, 10)`? What is produced by `expand(3, 8, 10)`?

Exercise 3.59

In section 2.5.3 we saw how to implement a polynomial arithmetic system representing polynomials as lists of terms. In a similar way, we can work with *power series*, such as

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \dots,$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \dots,$$

$$\sin x = x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \dots,$$

represented as infinite streams. We will represent the series $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$ as the stream whose elements are the coefficients $a_0, a_1, a_2, a_3, \dots$.

- a. The integral of the series $a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$ is the series

$$c + a_0x + \frac{1}{2}a_1x^2 + \frac{1}{3}a_2x^3 + \frac{1}{4}a_3x^4 + \dots$$

where c is any constant. Define a function `integrate_series` that takes as input a stream a_0, a_1, a_2, \dots representing a power series and returns the stream $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \dots$ of coefficients of the non-constant terms of the integral of the series. (Since the result has

⁶⁰This exercise shows how call-by-need is closely related to ordinary memoization as described in exercise 3.27. In that exercise, we used assignment to explicitly construct a local table. Our call-by-need stream optimization effectively constructs such a table automatically, storing values in the previously forced parts of the stream.

no constant term, it doesn't represent a power series; when we use `integrate-series`, we will pair with the appropriate constant.)

- b. The function $x \mapsto e^x$ is its own derivative. This implies that e^x and the integral of e^x are the same series, except for the constant term, which is $e^0 = 1$. Accordingly, we can generate the series for e^x as

```
const exp_series =
  pair(1, () => integrate_series(exp_series));
```

Show how to generate the series for sine and cosine, starting from the facts that the derivative of sine is cosine and the derivative of cosine is the negative of sine:

```
const cosine_series = pair(1, ??);
const sine_series = pair(0, ??);
```

Exercise 3.60

With power series represented as streams of coefficients as in exercise 3.59, adding series is implemented by `add-streams`. Complete the definition of the following function for multiplying series:

```
function mul_series(s1, s2) {
  pair(??, add_streams(??, ??));
}
```

You can test your function by verifying that $\sin^2 x + \cos^2 x = 1$, using the series from exercise 3.59.

Exercise 3.61

Let S be a power series (exercise 3.59) whose constant term is 1. Suppose we want to find the power series $1/S$, that is, the series X such that $S \cdot X = 1$. Write $S = 1 + S_R$ where S_R is the part of S after the constant term. Then we can solve for X as follows:

$$\begin{aligned} S \cdot X &= 1 \\ (1 + S_R) \cdot X &= 1 \\ X + S_R \cdot X &= 1 \\ X &= 1 - S_R \cdot X \end{aligned}$$

In other words, X is the power series whose constant term is 1 and whose higher-order terms are given by the negative of S_R times X . Use this idea to write a function `invert_unit_series` that computes $1/S$ for a power series S with constant term 1. You will need to use `mul_series` from exercise 3.60.

Exercise 3.62

Use the results of exercises 3.60 and 3.61 to define a function `div_series` that divides two power series. The function `div_series` should work for any two series, provided that the denominator series begins with a nonzero constant term. (If the denominator has a zero constant term, then `div_series` should signal an error.) Show how to use `div_series` together with the result of exercise 3.59 to generate the power series for tangent.

3.5.3 Exploiting the Stream Paradigm

Streams with delayed evaluation can be a powerful modeling tool, providing many of the benefits of local state and assignment. Moreover, they avoid some of the theoretical tangles that accompany the introduction of assignment into a programming language.

The stream approach can be illuminating because it allows us to build systems with different module boundaries than systems organized around assignment to state variables. For example, we can think of an entire time series (or signal) as a focus of interest, rather than the values of the state variables at individual moments. This makes it convenient to combine and compare components of state from different moments.

Formulating iterations as stream processes

In section 1.2.1, we introduced iterative processes, which proceed by updating state variables. We know now that we can represent state as a ‘timeless’ stream of values rather than as a set of variables to be updated. Let’s adopt this perspective in revisiting the square-root function from section 1.1.7. Recall that the idea is to generate a sequence of better and better guesses for the square root of x by applying over and over again the function that improves guesses:

```
function sqrt_improve(guess, x) {
    return average(guess, x / guess);
}
```

In our original `sqrt` function, we made these guesses be the successive values of a state variable. Instead we can generate the infinite stream of guesses, starting with an initial guess of 1:⁶¹

```
function sqrt_stream(x) {
    const guesses =
        pair(1.0,
            () => stream_map(guess => sqrt_improve(guess, x),
                guesses));
}
```

⁶¹We can’t use `let` to bind the local variable `guesses`, because the value of `guesses` depends on `guesses` itself. Exercise 3.63 addresses why we want a local variable here.

```

    return guesses;
}
display(eval_stream(sqrt_stream(2), 5));
// [1, [1.5, [1.4166666666666665, [1.4142156862745097,
// [1.4142135623746899, null]]]]]

```

We can generate more and more terms of the stream to get better and better guesses. If we like, we can write a function that keeps generating terms until the answer is good enough. (See exercise 3.64.)

Another iteration that we can treat in the same way is to generate an approximation to π , based upon the alternating series that we saw in section 1.3.1:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

We first generate the stream of summands of the series (the reciprocals of the odd integers, with alternating signs). Then we take the stream of sums of more and more terms (using the `partial_sums` function of exercise 3.55) and scale the result by 4:

```

function pi_summands(n) {
    return pair(1.0 / n,
        () => stream_map(x => -x,
            pi_summands(n + 2))
    );
}

const pi_stream =
    scale_stream(partial_sums(pi_summands(1)), 4);
display_stream(eval_stream(pi_stream, 8));
// [4, [2.6666666666666667, [3.4666666666666667,
// [2.8952380952380956, [3.3396825396825403,
// [2.9760461760461765, [3.2837384837384844,
// [3.017071817071818, null]]]]]]

```

This gives us a stream of better and better approximations to π , although the approximations converge rather slowly. Eight terms of the sequence bound the value of π between 3.284 and 3.017.

So far, our use of the stream of states approach is not much different from updating state variables. But streams give us an opportunity to do some interesting tricks. For example, we can transform a stream with a *sequence accelerator* that converts a sequence of approximations to a new sequence that converges to the same value as the original, only faster.

One such accelerator, due to the eighteenth-century Swiss mathematician Leonhard Euler, works well with sequences that are partial sums of alternating series (series of terms with alternating signs). In Euler's technique, if S_n is the n th term of the original sum sequence, then

the accelerated sequence has terms

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Thus, if the original sequence is represented as a stream of values, the transformed sequence is given by

```
function euler_transform(s) {
  const s0 = stream_ref(s, 0);
  const s1 = stream_ref(s, 1);
  const s2 = stream_ref(s, 2);
  return pair(s2 -
    square(s2 - s1) / (s0 + (-2) * s1 + s2),
    () => euler_transform(stream_tail(s)));
}
```

We can demonstrate Euler acceleration with our sequence of approximations to π :

```
display_stream(euler_transform(pi_stream));
// 3.1666666666666667
// 3.1333333333333337
// 3.1452380952380956
// 3.13968253968254
// 3.1427128427128435
// 3.1408813408813416
// 3.142071817071818
// 3.1412548236077655
// ...
```

Even better, we can accelerate the accelerated sequence, and recursively accelerate that, and so on. Namely, we create a stream of streams (a structure we'll call a *tableau*) in which each stream is the transform of the preceding one:

```
function make_tableau(transform, s) {
  return pair(s, () => make_tableau(transform, transform(s)));
}
```

The tableau has the form

s_{00}	s_{01}	s_{02}	s_{03}	s_{04}	...
s_{10}	s_{11}	s_{12}	s_{13}	...	
s_{20}	s_{21}	s_{22}	...		
...					

Finally, we form a sequence by taking the first term in each row of the tableau:

```
function accelerated_sequence(transform, s) {
  return stream_map(head, make_tableau(transform, s));
}
```

We can demonstrate this kind of ‘super-acceleration’ of the π sequence:

```
display(eval_stream(accelerated_sequence(euler_transform,
                                         pi_stream),
                    8));
// [4, [3.166666666666667, [3.142105263157895,
// [3.141599357319005, [3.1415927140337785, [3.1415926539752927,
// [3.1415926535911765, [3.141592653589778, null]]]]]]]
```

The result is impressive. Taking eight terms of the sequence yields the correct value of π to 14 decimal places. If we had used only the original π sequence, we would need to compute on the order of 10^{13} terms (i.e., expanding the series far enough so that the individual terms are less than 10^{-13}) to get that much accuracy!

We could have implemented these acceleration techniques without using streams. But the stream formulation is particularly elegant and convenient because the entire sequence of states is available to us as a data structure that can be manipulated with a uniform set of operations.

Exercise 3.63

Louis Reasoner asks why the `sqrt_stream` function was not written in the following more straightforward way, without the local variable `guesses`:

```
function sqrt_stream(x) {
    return pair(1.0,
               () => stream_map(guess =>
                               sqrt_improve(guess, x),
                               sqrt_stream(x))
               );
}
```

Alyssa P. Hacker replies that this version of the function is considerably less efficient because it performs redundant computation. Explain Alyssa’s answer. Would the two versions still differ in efficiency if our implementation of `delay` used only `???` without using the optimization provided by `memo-proc` (section 3.5.1)?

Exercise 3.64

Write a function `stream_limit` that takes as arguments a stream and a number (the tolerance). It should examine the stream until it finds two successive elements that differ in absolute value by less than the tolerance, and return the second of the two elements. Using this, we could compute square roots up to a given tolerance by

```
function sqrt(x, tolerance) {
```

```

    return stream_limit(sqrt_stream(x), tolerance);
}

```

Exercise 3.65

Use the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

to compute three sequences of approximations to the natural logarithm of 2, in the same way we did above for π . How rapidly do these sequences converge?

Infinite streams of pairs

In section 2.2.3, we saw how the sequence paradigm handles traditional nested loops as processes defined on sequences of pairs. If we generalize this technique to infinite streams, then we can write programs that are not easily represented as loops, because the ‘looping’ must range over an infinite set.

For example, suppose we want to generalize the prime_sum_pairs function of section 2.2.3 to produce the stream of pairs of *all* integers (i, j) with $i \leq j$ such that $i + j$ is prime. If int_pairs is the sequence of all pairs of integers (i, j) with $i \leq j$, then our required stream is simply⁶²

```

stream_filter(pair => is_prime(head(pair) + head(tail(pair))),
              int_pairs);

```

Our problem, then, is to produce the stream int_pairs. More generally, suppose we have two streams $S = (S_i)$ and $T = (T_j)$, and imagine the infinite rectangular array

$$\begin{array}{ccccccc}
 (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \dots \\
 (S_1, T_0) & (S_1, T_1) & (S_1, T_2) & \dots \\
 (S_2, T_0) & (S_2, T_1) & (S_2, T_2) & \dots \\
 & \dots & & & & & \\
 \end{array}$$

We wish to generate a stream that contains all the pairs in the array that lie on or above the diagonal, i.e., the pairs

$$\begin{array}{ccccccc}
 (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \dots \\
 (S_1, T_1) & (S_1, T_2) & \dots \\
 (S_2, T_2) & \dots \\
 & \dots & & & & & \\
 \end{array}$$

(If we take both S and T to be the stream of integers, then this will be our desired stream int_pairs.)

⁶²As in section 2.2.3, we represent a pair of integers as a list rather than a JavaScript pair.

Call the general stream of pairs `pairs(s, t)`, and consider it to be composed of three parts: the pair (S_0, T_0) , the rest of the pairs in the first row, and the remaining pairs.⁶³

(S_0, T_0)	(S_0, T_1)	(S_0, T_2)	\dots
	(S_1, T_1)	(S_1, T_2)	\dots
		(S_2, T_2)	\dots
			\dots

Observe that the third piece in this decomposition (pairs that are not in the first row) is (recursively) the pairs formed from `stream_tail(s)` and `stream_tail(t)`. Also note that the second piece (the rest of the first row) is

```
stream_map(x => list(head(s), x),
            stream_tail(t));
```

Thus we can form our stream of pairs as follows:

```
function pairs(s, t) {
    return pair(list(head(s), head(t)),
                () => <combine_in_some_way>(
                    stream_map(x => list(head(s), x),
                               stream_tail(t)),
                    pairs(stream_tail(s), stream_tail(t)))
                );
}
```

In order to complete the function, we must choose some way to combine the two inner streams. One idea is to use the stream analog of the `append` function from section 2.2.1:

```
function stream_append(s1, s2) {
    return is_null(s1)
        ? s2
        : pair(head(s1),
               () => stream_append(stream_tail(s1), s2)
               );
}
```

This is unsuitable for infinite streams, however, because it takes all the elements from the first stream before incorporating the second stream. In particular, if we try to generate all pairs of positive integers using

```
pairs(integers, integers);
```

our stream of results will first try to run through all pairs with the first integer equal to 1, and hence will never produce pairs with any other value of the first integer.

⁶³See exercise 3.68 for some insight into why we chose this decomposition.

To handle infinite streams, we need to devise an order of combination that ensures that every element will eventually be reached if we let our program run long enough. An elegant way to accomplish this is with the following `interleave` function:⁶⁴

```
function interleave(s1, s2) {
    return is_null(s1)
        ? s2;
        : pair(head(s1),
            () => interleave(s2, stream_tail(s1)))
        );
}
```

Since `interleave` takes elements alternately from the two streams, every element of the second stream will eventually find its way into the interleaved stream, even if the first stream is infinite.

We can thus generate the required stream of pairs as

```
function pairs(s, t) {
    return pair(list(head(s), head(t)),
        () => interleave(stream_map(x => list(head(s),
            x),
            stream_tail(t)),
        pairs(stream_tail(s),
            stream_tail(t))));
}
```

Exercise 3.66

Examine the stream `pairs(integers, integers)`. Can you make any general comments about the order in which the pairs are placed into the stream? For example, about how many pairs precede the pair (1,100)? the pair (99,100)? the pair (100,100)? (If you can make precise mathematical statements here, all the better. But feel free to give more qualitative answers if you find yourself getting bogged down.)

Exercise 3.67

Modify the `pairs` function so that `pairs(integers, integers)` will produce the stream of *all* pairs of integers (i, j) (without the condition $i \leq j$). Hint: You will need to mix in an additional stream.

Exercise 3.68

⁶⁴The precise statement of the required property on the order of combination is as follows: There should be a function f of two arguments such that the pair corresponding to element i of the first stream and element j of the second stream will appear as element number $f(i, j)$ of the output stream. The trick of using `interleave` to accomplish this was shown to us by David Turner, who employed it in the language KRC (Turner 1981).

Louis Reasoner thinks that building a stream of pairs from three parts is unnecessarily complicated. Instead of separating the pair (S_0, T_0) from the rest of the pairs in the first row, he proposes to work with the whole first row, as follows:

```
function pairs(s, t) {
    return interleave(stream_map(x => list(head(s), x),
                                t),
                    pair(stream_tail(s), stream_tail(t)));
}
```

Does this work? Consider what happens if we evaluate `pairs(integers, integers)` using Louis's definition of `pairs`.

Exercise 3.69

Write a function `triples` that takes three infinite streams, S , T , and U , and produces the stream of triples (S_i, T_j, U_k) such that $i \leq j \leq k$. Use `triples` to generate the stream of all Pythagorean triples of positive integers, i.e., the triples (i, j, k) such that $i \leq j$ and $i^2 + j^2 = k^2$.

Exercise 3.70

It would be nice to be able to generate streams in which the pairs appear in some useful order, rather than in the order that results from an *ad hoc* interleaving process. We can use a technique similar to the `merge` function of exercise 3.56, if we define a way to say that one pair of integers is ‘less than’ another. One way to do this is to define a ‘weighting function’ $W(i, j)$ and stipulate that (i_1, j_1) is less than (i_2, j_2) if $W(i_1, j_1) < W(i_2, j_2)$. Write a function `merge_weighted` that is like `merge`, except that `merge_weighted` takes an additional argument `weight`, which is a function that computes the weight of a pair, and is used to determine the order in which elements should appear in the resulting merged stream.⁶⁵ Using this, generalize `pairs` to a function `weighted_pairs` that takes two streams, together with a function that computes a weighting function, and generates the stream of pairs, ordered according to `weight`. Use your function to generate

- the stream of all pairs of positive integers (i, j) with $i \leq j$ ordered according to the sum $i + j$
- the stream of all pairs of positive integers (i, j) with $i \leq j$, where neither i nor j is divisible by 2, 3, or 5, and the pairs are ordered according to the sum $2i + 3j + 5ij$.

⁶⁵We will require that the weighting function be such that the weight of a pair increases as we move out along a row or down along a column of the array of pairs.

Exercise 3.71

Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called *Ramanujan numbers*, in honor of the mathematician Srinivasa Ramanujan.⁶⁶ Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$ (see exercise 3.70), then search the stream for two consecutive pairs with the same weight. Write a function to generate the Ramanujan numbers. The first such number is 1,729. What are the next five?

Exercise 3.72

In a similar way to exercise 3.71 generate a stream of all numbers that can be written as the sum of two squares in three different ways (showing how they can be so written).

Streams as signals

We began our discussion of streams by describing them as computational analogs of the ‘signals’ in signal-processing systems. In fact, we can use streams to model signal-processing systems in a very direct way, representing the values of a signal at successive time intervals as consecutive elements of a stream. For instance, we can implement an *integrator* or *summer* that, for an input stream $x = (x_i)$, an initial value C , and a small increment dt , accumulates the sum

$$S_i = C + \sum_{j=1}^i x_j dt$$

and returns the stream of values $S = (S_i)$. The following `integral` function is reminiscent of the ‘implicit style’ definition of the stream of integers (section 3.5.2):

```
function integral(integrand, initial_value, dt) {
  const integ = pair(initial_value,
    () => add_streams(scale_stream(integrand, dt),
      integ));
  return integ;
}
```

⁶⁶To quote from G. H. Hardy’s obituary of Ramanujan (Hardy 1921): ‘It was Mr. Littlewood (I believe) who remarked that “every positive integer was one of his friends.” I remember once going to see him when he was lying ill at Putney. I had ridden in taxi-cab No. 1729, and remarked that the number seemed to me a rather dull one, and that I hoped it was not an unfavorable omen. “No,” he replied, “it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.” The trick of using weighted pairs to generate the Ramanujan numbers was shown to us by Charles Leiserson.

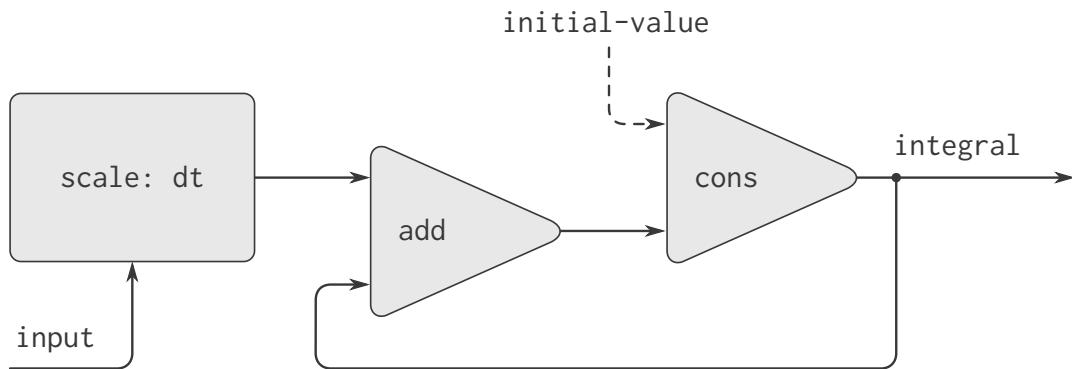


Figure 3.32: The integral function viewed as a signal-processing system.

Figure 3.32 is a picture of a signal-processing system that corresponds to the `integral` function. The input stream is scaled by dt and passed through an adder, whose output is passed back through the same adder. The self-reference in the definition of `int` is reflected in the figure by the feedback loop that connects the output of the adder to one of the inputs.

Exercise 3.73

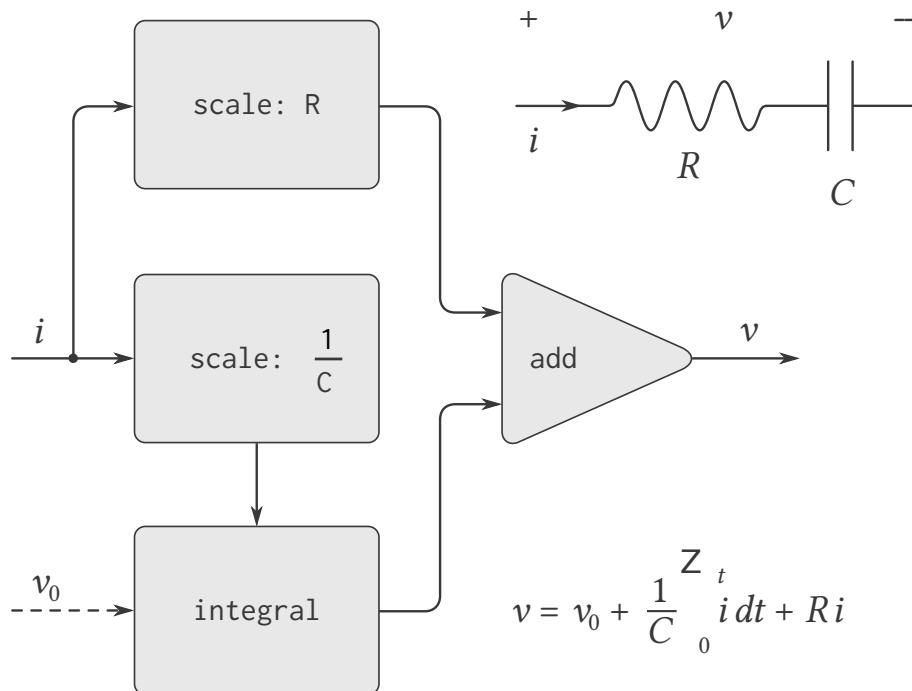


Figure 3.33: An RC circuit and the associated signal-flow diagram.

We can model electrical circuits using streams to represent the values of currents or voltages at a sequence of times. For instance, suppose we have an *RC circuit* consisting of a resistor of resistance R and a capacitor of capacitance C in series. The voltage response v of the circuit to an injected current i is determined by the formula in Figure 3.33, whose structure is shown by the accompanying signal-flow diagram.

Write a function `RC` that models this circuit. `RC` should take as inputs the values of R , C , and dt and should return a function that takes as inputs a stream representing the current i and an initial value for the capacitor voltage v_0 and produces as output the stream of voltages v . For example, you should be able to use `RC` to model an RC circuit with $R = 5$ ohms, $C = 1$ farad, and a 0.5-second time step by evaluating `const RC1 = RC(5, 1, 0.5)`. This defines `RC1` as a function that takes a stream representing the time sequence of currents and an initial capacitor voltage and produces the output stream of voltages.

Exercise 3.74

Alyssa P. Hacker is designing a system to process signals coming from physical sensors. One important feature she wishes to produce is a signal that describes the *zero crossings* of the input signal. That is, the resulting signal should be $+1$ whenever the input signal changes from negative to positive, -1 whenever the input signal changes from positive to negative, and 0 otherwise. (Assume that the sign of a 0 input is positive.) For example, a typical input signal with its associated zero-crossing signal would be

```
... 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 ...
... 0 0 0 0 -1 0 0 0 0 1 0 0 ...
```

In Alyssa's system, the signal from the sensor is represented as a stream `sense_data` and the stream `zero_crossings` is the corresponding stream of zero crossings. Alyssa first writes a function `sign_change_detector` that takes two values as arguments and compares the signs of the values to produce an appropriate 0 , 1 , or -1 . She then constructs her zero-crossing stream as follows:

```
function make_zero_crossings(input_stream, last_value) {
    return pair(sign_change_detector(head(input_stream),
                                      last_value),
               () => make_zero_crossings(
                   stream_tail(input_stream),
                   head(input_stream)));
}
const zero_crossings = make_zero_crossings(sense_data, 0);
```

Alyssa's boss, Eva Lu Ator, walks by and suggests that this program is approximately equivalent to the following one, which uses the function `combine_streams` from exercise 3.50:

```
const zero_crossing = combine_streams(sign_change_detector,
                                         sense_data,
                                         <expression>);
```

Complete the program by supplying the indicated `<expression>`.

Exercise 3.75

Unfortunately, Alyssa's zero-crossing detector in exercise 3.74 proves to be insufficient, because the noisy signal from the sensor leads to spurious zero crossings. Lem E. Tweakit, a hardware specialist, suggests that Alyssa smooth the signal to filter out the noise before extracting the zero crossings. Alyssa takes his advice and decides to extract the zero crossings from the signal constructed by averaging each value of the sense data with the previous value. She explains the problem to her assistant, Louis Reasoner, who attempts to implement the idea, altering Alyssa's program as follows:

```
function make_zero_crossings(input_stream, last_value) {
  const avpt = (head(input_stream) + last_value) / 2;
  return pair(sign_change_detector(avpt, last_value),
    () => make_zero_crossings(
      stream_tail(input_stream),
      avpt));
}
```

This does not correctly implement Alyssa's plan. Find the bug that Louis has installed and fix it without changing the structure of the program. (Hint: You will need to increase the number of arguments to `make_zero_crossings`.)

Exercise 3.76

Eva Lu Ator has a criticism of Louis's approach in exercise 3.75. The program he wrote is not modular, because it intermixes the operation of smoothing with the zero-crossing extraction. For example, the extractor should not have to be changed if Alyssa finds a better way to condition her input signal. Help Louis by writing a function `smooth` that takes a stream as input and produces a stream in which each element is the average of two successive input stream elements. Then use `smooth` as a component to implement the zero-crossing detector in a more modular style.

3.5.4 Streams and Delayed Evaluation

The `integral` function at the end of the preceding section shows how we can use streams to model signal-processing systems that contain feedback loops. The feedback loop for the adder shown in figure 3.32 is modeled by the fact that `integral`'s internal stream `int` is defined in terms of itself:

```
const integ = pair(initial_value,
  () => add_streams(scale_stream(integrand, dt),
    integ));
)
```

The interpreter's ability to deal with such an implicit definition depends on the delay resulting from wrapping the call of `add_streams` into a function definition. Without this delay, the interpreter could not construct `integ` before evaluating both arguments to `pair`, which would require that `integ` already be defined. In general, such a delay is crucial for using streams to model signal-processing systems that contain loops. Without a delay, our models would have to be formulated so that the inputs to any signal-processing component would be fully evaluated before the output could be produced. This would outlaw loops.

Unfortunately, stream models of systems with loops may require uses of a delay beyond the stream programming pattern seen so far. For instance, figure 3.34 shows a signal-processing system for solving the differential equation $dy/dt = f(y)$ where f is a given function. The figure shows a mapping component, which applies f to its input signal, linked in a feedback loop to an integrator in a manner very similar to that of the analog computer circuits that are actually used to solve such equations.

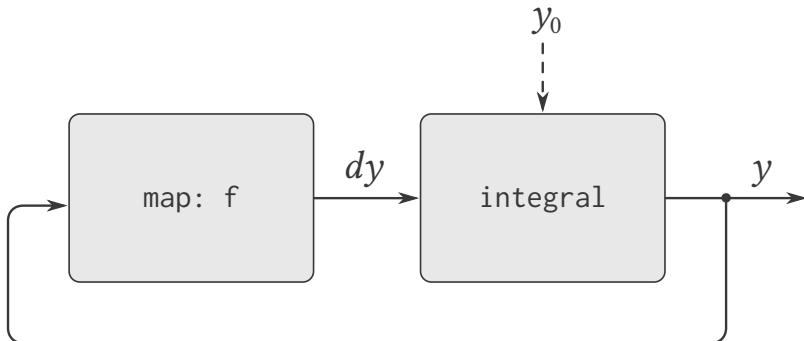


Figure 3.34: An ‘analog computer circuit’ that solves the equation

Assuming we are given an initial value y_0 for y , we could try to model this system using the function

```

function solve(f, y0, dt) {
  const y = integral(dy, y0, dt);
  const dy = stream_map(f, y);
  return y;
}
  
```

This function does not work, because in the first line of `solve` the call to `integral` requires that the input `dy` be defined, which does not happen until the second line of `solve`.

On the other hand, the intent of our definition does make sense, because we can, in principle, begin to generate the `y` stream without knowing `dy`. Indeed, `integral` and many other stream operations can generate part of the answer given only partial information about the arguments. For `integral`, the first element of the output stream is the specified `initial_value`. Thus, we can generate the first element of the output stream without evaluating the integrand `dy`. Once we know the first element of `y`, the `stream_map` in the second line of `solve` can begin working

to generate the first element of dy , which will produce the next element of y , and so on.

To take advantage of this idea, we will redefine `integral` to expect the integrand stream to be a *delayed argument*. The function `integral` will force the integrand to be evaluated only when it is required to generate more than the first element of the output stream:

```
function integral(delayed_integrand, intial_value, dt) {
    const integrand = delayed_integrand();
    const integ =
        pair(intial_value,
            add_streams(scale_stream(integrand, dt), int));
}
```

Now we can implement our `solve` function by delaying the evaluation of dy in the definition of y :

```
function solve(f, y0, dt) {
    const y = integral( () => dy, y0, dt);
    const dy = stream_map(f, y);
    return y;
}
```

In general, every caller of `integral` must now delay the integrand argument. We can demonstrate that the `solve` function works by approximating $e \approx 2.718$ by computing the value at $y = 1$ of the solution to the differential equation $dy/dt = y$ with initial condition $y(0) = 1$:

```
stream_ref(solve(y => y, 1, 0.001), 1000);
```

Exercise 3.77

The `integral` function used above was analogous to the ‘implicit’ definition of the infinite stream of integers in section 3.5.2. Alternatively, we can give a definition of `integral` that is more like `integers-starting-from` (also in section 3.5.2):

```
function integral(integrand, intial_value, dt) {
    return pair(intial_value,
        is_null(integrand) ? null
            : integral(stream_tail(integrand),
                dt * head(integrand) + intial_value,
                dt));
}
```

When used in systems with loops, this function has the same problem as does our original version of `integral`. Modify the function so that it expects the `integrand` as a delayed argument and hence can be used in the `solve` function shown above.

Exercise 3.78

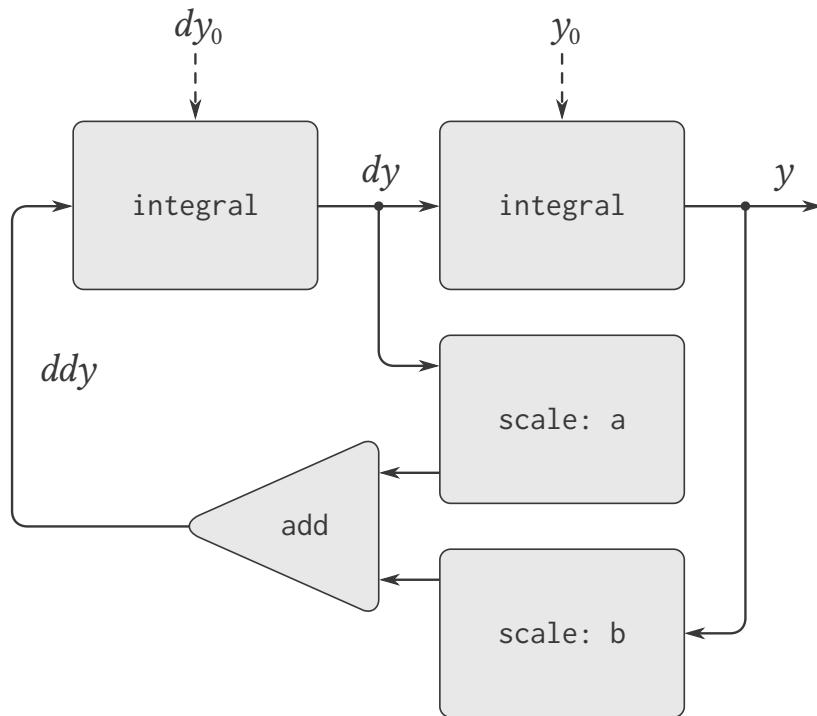


Figure 3.35: Signal-flow diagram for the solution to a second-order linear differential equation.

Consider the problem of designing a signal-processing system to study the homogeneous second-order linear differential equation

$$\frac{d^2y}{dt^2} - a\frac{dy}{dt} - by = 0$$

The output stream, modeling y , is generated by a network that contains a loop. This is because the value of d^2y/dt^2 depends upon the values of y and dy/dt and both of these are determined by integrating d^2y/dt^2 . The diagram we would like to encode is shown in Figure 3.35. Write a function `solve_2nd` that takes as arguments the constants a , b , and dt and the initial values y_0 and dy_0 for y and dy/dt and generates the stream of successive values of y .

Exercise 3.79

Generalize the `solve-2nd` function of exercise 3.78 so that it can be used to solve general second-order differential equations $d^2y/dt^2 = f(dy/dt, y)$.

Exercise 3.80

A *series RLC circuit* consists of a resistor, a capacitor, and an inductor connected in series, as shown in Figure 3.36. If R , L , and C are the resistance, inductance, and capacitance, then the relations between voltage (v) and current (i) for the three components are described by the

equations

$$\begin{aligned} v_R &= i_R R \\ v_L &= L \frac{di_L}{dt} \\ i_C &= C \frac{dv_C}{dt} \end{aligned}$$

and the circuit connections dictate the relations

$$\begin{aligned} i_R &= i_L = -i_C \\ v_C &= v_L + v_R \end{aligned}$$

Combining these equations shows that the state of the circuit (summarized by v_C , the voltage across the capacitor, and i_L , the current in the inductor) is described by the pair of differential equations

$$\begin{aligned} \frac{dv_C}{dt} &= -\frac{i_L}{C} \\ \frac{di_L}{dt} &= \frac{1}{L}v_C - \frac{R}{L}i_L \end{aligned}$$

The signal-flow diagram representing this system of differential equations is shown in Figure 3.37.

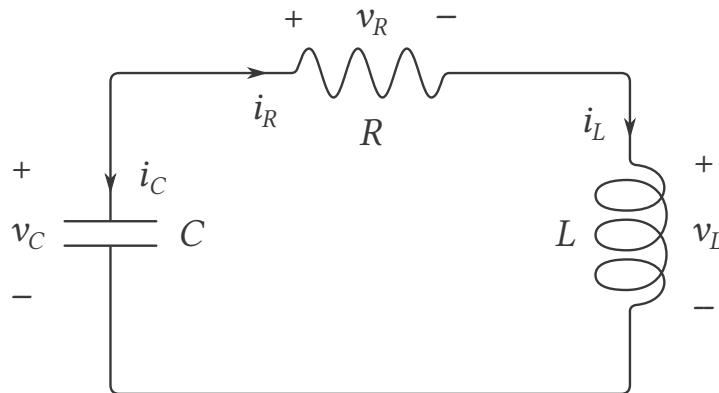


Figure 3.36: A series RLC circuit.

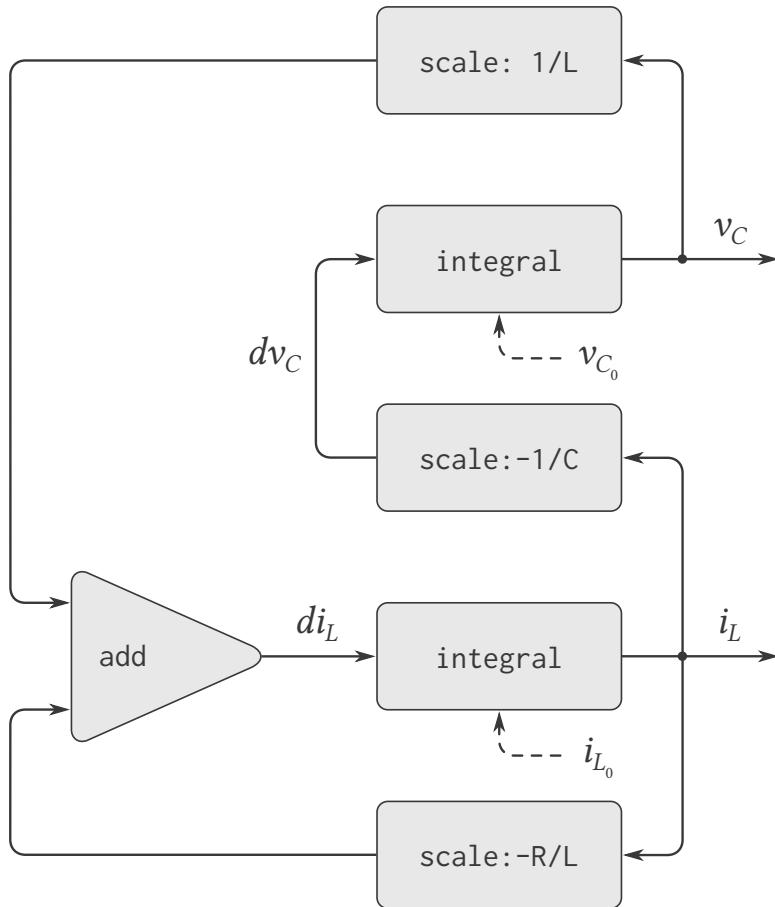


Figure 3.37: A signal-flow diagram for the solution to a series RLC circuit.

Write a function `RLC` that takes as arguments the parameters R , L , and C of the circuit and the time increment dt . In a manner similar to that of the `RC` function of exercise 3.73, `RLC` should produce a function that takes the initial values of the state variables, v_{C_0} and i_{L_0} , and produces a pair (using `pair`) of the streams of states v_C and i_L . Using `RLC`, generate the pair of streams that models the behavior of a series RLC circuit with $R = 1$ ohm, $C = 0.2$ farad, $L = 1$ henry, $dt = 0.1$ second, and initial values $i_{L_0} = 0$ amps and $v_{C_0} = 10$ volts.

Normal-order evaluation

The examples in this section illustrate how delayed evaluation provides great programming flexibility, but the same examples also show how this can make our programs more complex. Our new `integral` function, for instance, gives us the power to model systems with loops, but we must now remember that `integral` should be called with a delayed integrand, and every function that uses `integral` must be aware of this. In effect, we have created two classes of functions: ordinary functions and functions that take delayed arguments. In general, creating separate classes of functions forces us to create separate classes of higher-order functions as

well.⁶⁷

One way to avoid the need for two different classes of functions is to make all functions take delayed arguments. We could adopt a model of evaluation in which all arguments to functions are automatically delayed and arguments are forced only when they are actually needed (for example, when they are required by a primitive operation). This would transform our language to use normal-order evaluation, which we first described when we introduced the substitution model for evaluation in section 1.1.5. Converting to normal-order evaluation provides a uniform and elegant way to simplify the use of delayed evaluation, and this would be a natural strategy to adopt if we were concerned only with stream processing. In section 4.2, after we have studied the evaluator, we will see how to transform our language in just this way. Unfortunately, including delays in function calls wreaks havoc with our ability to design programs that depend on the order of events, such as programs that use assignment, mutate data, or perform input or output. Even a single delay in the tail of a pair can cause great confusion, as illustrated by exercise 3.51 and 3.52. As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, and devising ways to deal with both of these at once is an active area of research.

3.5.5 Modularity of Functional Programs and Modularity of Objects

As we saw in section 3.1.2, one of the major benefits of introducing assignment is that we can increase the modularity of our systems by encapsulating, or ‘hiding,’ parts of the state of a large system within local variables. Stream models can provide an equivalent modularity without the use of assignment. As an illustration, we can reimplement the Monte Carlo estimation of π , which we examined in section 3.1.2, from a stream-processing point of view.

The key modularity issue was that we wished to hide the internal state of a random-number generator from programs that used random numbers. We began with a function `rand_update`, whose successive values furnished our supply of random numbers, and used this to produce a random-number generator:

⁶⁷This is a small reflection, in JavaScript, of the difficulties that conventional strongly typed languages such as Pascal have in coping with higher-order functions. In such languages, the programmer must specify the data types of the arguments and the result of each function: number, logical value, sequence, and so on. Consequently, we could not express an abstraction such as ‘map a given function `fun` over all the elements in a sequence’ by a single higher-order function such as `stream_map`. Rather, we would need a different mapping function for each different combination of argument and result data types that might be specified for a `fun`. Maintaining a practical notion of ‘data type’ in the presence of higher-order functions raises many difficult issues. One way of dealing with this problem is illustrated by the language ML (Gordon, Milner, and Wadsworth 1979), whose ‘polymorphic data types’ include templates for higher-order transformations between data types. Moreover, data types for most functions in ML are never explicitly declared by the programmer. Instead, ML includes a *type-inferencing* mechanism that uses information in the environment to deduce the data types for newly defined functions.

```
function make_rand() {
  let x = random_init;
  function rand() {
    x = rand_update(x);
    return x;
  }
  return rand;
}
const rand = make_rand();
```

In the stream formulation there is no random-number generator *per se*, just a stream of random numbers produced by successive calls to `rand_update`:

```
const random_numbers =
  pair(random_init,
    () => stream_map(rand_update, random_numbers));
```

We use this to construct the stream of outcomes of the Cesàro experiment performed on consecutive pairs in the `random_numbers` stream:

```
function map_successive_pairs(f, s) {
  return pair(f(head(s), head(stream_tail(s))),
    map_successive_pairs(
      f,
      stream_tail(stream_tail(s))));
}
const cesaro_stream =
  map_successive_pairs( (r1, r2) => gcd(r1, r2) === 1,
    random_numbers);
```

The `cesaro_stream` is now fed to a `monte_carlo` function, which produces a stream of estimates of probabilities. The results are then converted into a stream of estimates of π . This version of the program doesn't need a parameter telling how many trials to perform. Better estimates of π (from performing more experiments) are obtained by looking farther into the `pi` stream:

```
function monte_carlo(experiment_stream, passed, failed) {
  function next(passed, failed) {
    return pair(passed / (passed + failed),
      monte_carlo(stream_tail(experiment_stream),
        passed, failed));
  }
  return head(experiment_stream)
    ? next(passed + 1, failed)
    : next(passed, failed + 1);
}

const pi = stream_map(p => sqrt(6 / p),
  monte_carlo(cesaro_stream(0, 0)));
```

There is considerable modularity in this approach, because we still can formulate a general `monte_carlo` function that can deal with arbitrary experiments. Yet there is no assignment or local state.

Exercise 3.81

Exercise 3.6 discussed generalizing the random-number generator to allow one to reset the random-number sequence so as to produce repeatable sequences of ‘random’ numbers. Produce a stream formulation of this same generator that operates on an input stream of requests to generate a new random number or to reset the sequence to a specified value and that produces the desired stream of random numbers. Don’t use assignment in your solution.

Exercise 3.82

Redo exercise 3.5 on Monte Carlo integration in terms of streams. The stream version of `estimate_integral` will not have an argument telling how many trials to perform. Instead, it will produce a stream of estimates based on successively more trials.

A functional-programming view of time

Let us now return to the issues of objects and state that were raised at the beginning of this chapter and examine them in a new light. We introduced assignment and mutable objects to provide a mechanism for modular construction of programs that model systems with state. We constructed computational objects with local state variables and used assignment to modify these variables. We modeled the temporal behavior of the objects in the world by the temporal behavior of the corresponding computational objects.

Now we have seen that streams provide an alternative way to model objects with local state. We can model a changing quantity, such as the local state of some object, using a stream that represents the time history of successive states. In essence, we represent time explicitly, using streams, so that we decouple time in our simulated world from the sequence of events that take place during evaluation. Indeed, because of the presence of `delay` there may be little relation between simulated time in the model and the order of events during the evaluation.

In order to contrast these two approaches to modeling, let us reconsider the implementation of a ‘withdrawal processor’ that monitors the balance in a bank account. In section 3.1.3 we implemented a simplified version of such a processor:

```
function make_simplified_withdraw(balance) {
    function withdraw(amount) {
        balance = balance - amount;
        return balance;
    }
}
```

```

    }
    return withdraw;
}

```

Calls to `make_simplified_withdraw` produce computational objects, each with a local state variable `balance` that is decremented by successive calls to the object. The object takes an amount as an argument and returns the new balance. We can imagine the user of a bank account typing a sequence of inputs to such an object and observing the sequence of returned values shown on a display screen.

Alternatively, we can model a withdrawal processor as a function that takes as input a balance and a stream of amounts to withdraw and produces the stream of successive balances in the account:

```

function stream_withdraw(balance, amount_stream) {
    return pair(balance,
        () => stream_withdraw(
            balance - head(amount_stream),
            stream_tail(amount_stream)));
}

```

The function `stream_withdraw` implements a well-defined mathematical function whose output is fully determined by its input. Suppose, however, that the input `amount_stream` is the stream of successive values typed by the user and that the resulting stream of balances is displayed. Then, from the perspective of the user who is typing values and watching results, the stream process has the same behavior as the object created by `make_simplified_withdraw`. However, with the stream version, there is no assignment, no local state variable, and consequently none of the theoretical difficulties that we encountered in section 3.1.3. Yet the system has state!

This is really remarkable. Even though `stream_withdraw` implements a well-defined mathematical function whose behavior does not change, the user's perception here is one of interacting with a system that has a changing state. One way to resolve this paradox is to realize that it is the user's temporal existence that imposes state on the system. If the user could step back from the interaction and think in terms of streams of balances rather than individual transactions, the system would appear stateless.⁶⁸

From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write programs that model this kind of natural decomposition in our world (as we see it from our viewpoint as a part of that world) with structures in our computer, we make computational objects that are not functional—they must change with time. We model state with local state variables, and we

⁶⁸Similarly in physics, when we observe a moving particle, we say that the position (state) of the particle is changing. However, from the perspective of the particle's world line in space-time there is no change involved.

model the changes of state with assignments to those variables. By doing this we make the time of execution of a computation model time in the world that we are part of, and thus we get ‘objects’ in our computer.

Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are part. However, as we’ve seen repeatedly throughout this chapter, these models raise thorny problems of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of *functional programming languages*, which do not include any provision for assignment or mutable data. In such a language, all functions implement well-defined mathematical functions of their arguments, whose behavior does not change. The functional approach is extremely attractive for dealing with concurrent systems.⁶⁹

On the other hand, if we look closely, we can see time-related problems creeping into functional models as well. One particularly troublesome area arises when we wish to design interactive systems, especially ones that model interactions between independent entities. For instance, consider once more the implementation a banking system that permits joint bank accounts. In a conventional system using assignment and objects, we would model the fact that Peter and Paul share an account by having both Peter and Paul send their transaction requests to the same bank-account object, as we saw in section 3.1.3. From the stream point of view, where there are no ‘objects’ *per se*, we have already indicated that a bank account can be modeled as a process that operates on a stream of transaction requests to produce a stream of responses. Accordingly, we could model the fact that Peter and Paul have a joint bank account by merging Peter’s stream of transaction requests with Paul’s stream of requests and feeding the result to the bank-account stream process, as shown in figure 3.38.

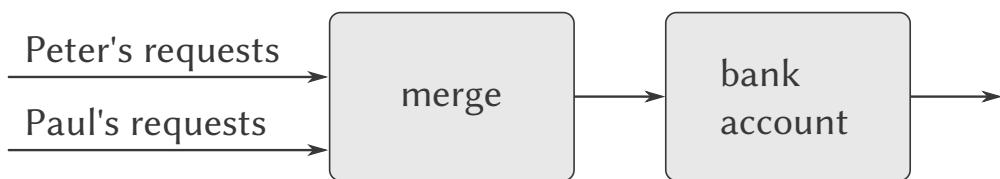


Figure 3.38: A joint bank account, modeled by merging two streams of transaction requests.

The trouble with this formulation is in the notion of *merge*. It will not do to merge the two streams by simply taking alternately one request from Peter and one request from Paul. Suppose Paul accesses the account only very rarely. We could hardly force Peter to wait for Paul to access the account before he could issue a second transaction. However such a merge is implemented, it must interleave the two transaction streams in some way that is constrained

⁶⁹John Backus, the inventor of Fortran, gave high visibility to functional programming when he was awarded the ACM Turing award in 1978. His acceptance speech (Backus 1978) strongly advocated the functional approach. A good overview of functional programming is given in Henderson 1980 and in Darlington, Henderson, and Turner 1982.

by ‘real time’ as perceived by Peter and Paul, in the sense that, if Peter and Paul meet, they can agree that certain transactions were processed before the meeting, and other transactions were processed after the meeting.⁷⁰

This is precisely the same constraint that we had to deal with in section 3.4.1, where we found the need to introduce explicit synchronization to ensure a ‘correct’ order of events in concurrent processing of objects with state. Thus, in an attempt to support the functional style, the need to merge inputs from different agents reintroduces the same problems that the functional style was meant to eliminate.

We began this chapter with the goal of building computational models whose structure matches our perception of the real world we are trying to model. We can model the world as a collection of separate, time-bound, interacting objects with state, or we can model the world as a single, timeless, stateless unity. Each view has powerful advantages, but neither view alone is completely satisfactory. A grand unification has yet to emerge.⁷¹

⁷⁰Observe that, for any two streams, there is in general more than one acceptable order of interleaving. Thus, technically, ‘merge’ is a relation rather than a function—the answer is not a deterministic function of the inputs. We already mentioned (footnote 38) that nondeterminism is essential when dealing with concurrency. The merge relation illustrates the same essential nondeterminism, from the functional perspective.

⁷¹The object model approximates the world by dividing it into separate pieces. The functional model does not modularize along object boundaries. The object model is useful when the unshared state of the ‘objects’ is much larger than the state that they share. An example of a place where the object viewpoint fails is quantum mechanics, where thinking of things as individual particles leads to paradoxes and confusions. Unifying the object view with the functional view may have little to do with programming, but rather with fundamental epistemological issues.

Chapter 4

Metalinguistic Abstraction

...It's in words that the magic is—Abracadabra, Open Sesame, and the rest—but the magic words in one story aren't magical in the next. The real magic is to understand which words work, and when, and for what; the trick is to learn the trick.

...And those words are made from the letters of our alphabet: a couple-dozen squiggles we can draw with the pen. This is the key! And the treasure, too, if we can only get our hands on it! It's as if—as if the key to the treasure *is* the treasure!

— John Barth, *Chimera*

In our study of program design, we have seen that expert programmers control the complexity of their designs with the same general techniques used by designers of all complex systems. They combine primitive elements to form compound objects, they abstract compound objects to form higher-level building blocks, and they preserve modularity by adopting appropriate large-scale views of system structure. In illustrating these techniques, we have used JavaScript as a language for describing processes and for constructing computational data objects and processes to model complex phenomena in the real world. However, as we confront increasingly complex problems, we will find that JavaScript, or indeed any fixed programming language, is not sufficient for our needs. We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.¹

¹The same idea is pervasive throughout all of engineering. For example, electrical engineers use many different languages for describing circuits. Two of these are the language of electrical *networks* and the language of

Programming is endowed with a multitude of languages. There are physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as function definition, that are appropriate to the larger-scale organization of systems.

Metalinguistic abstraction—establishing new languages—plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An *evaluator* (or *interpreter*) for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression.

It is no exaggeration to regard this as the most fundamental idea in programming:

The evaluator, which determines the meaning of expressions in a programming language, is just another program.

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

In fact, we can regard almost any program as the evaluator for some language. For instance, the polynomial manipulation system of section 2.5.3 embodies the rules of polynomial arithmetic and implements them in terms of operations on list-structured data. If we augment this system with functions to read and print polynomial expressions, we have the core of a special-purpose language for dealing with problems in symbolic mathematics. The digital-logic simulator of section 3.3.4 and the constraint propagator of section 3.3.5 are legitimate languages in their own right, each with its own primitives, means of combination, and means of abstraction. Seen from this perspective, the technology for coping with large-scale computer systems merges

electrical systems. The network language emphasizes the physical modeling of devices in terms of discrete electrical elements. The primitive objects of the network language are primitive electrical components such as resistors, capacitors, inductors, and transistors, which are characterized in terms of physical variables called voltage and current. When describing circuits in the network language, the engineer is concerned with the physical characteristics of a design. In contrast, the primitive objects of the system language are signal-processing modules such as filters and amplifiers. Only the functional behavior of the modules is relevant, and signals are manipulated without concern for their physical realization as voltages and currents. The system language is erected on the network language, in the sense that the elements of signal-processing systems are constructed from electrical networks. Here, however, the concerns are with the large-scale organization of electrical devices to solve a given application problem; the physical feasibility of the parts is assumed. This layered collection of languages is another example of the stratified design technique illustrated by the picture language of section 2.2.4.

with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages.

We now embark on a tour of the technology by which languages are established in terms of other languages. In this chapter we shall use JavaScript as a base, implementing evaluators as JavaScript functions. JavaScript is particularly well suited to this task, because of its ability to represent and manipulate symbolic expressions. We will take the first step in understanding how languages are implemented by building an evaluator for JavaScript itself. The language implemented by our evaluator will be a subset of JavaScript. Although the evaluator described in this chapter is written for a particular subset of JavaScript, it contains the essential structure of an evaluator for any expression-oriented language designed for writing programs for a sequential machine. (In fact, most language processors contain, deep within them, a little evaluator.) The evaluator has been simplified for the purposes of illustration and discussion, and some features have been left out that would be important to include in a production-quality JavaScript system. Nevertheless, this simple evaluator is adequate to execute most of the programs in this book.²

An important advantage of making the evaluator accessible as a JavaScript program is that we can implement alternative evaluation rules by describing these as modifications to the evaluator program. One place where we can use this power to good effect is to gain extra control over the ways in which computational models embody the notion of time, which was so central to the discussion in chapter 3. There, we mitigated some of the complexities of state and assignment by using streams to decouple the representation of time in the world from time in the computer. Our stream programs, however, were sometimes cumbersome, because they were constrained by the applicative-order evaluation of JavaScript. In section 4.2, we'll change the underlying language to provide for a more elegant approach, by modifying the evaluator to provide for *normal-order evaluation*.

4.1 The Metacircular Evaluator

Our evaluator for JavaScript will be implemented as a JavaScript program. It may seem circular to think about evaluating JavaScript programs using an evaluator that is itself implemented in JavaScript. However, evaluation is a process, so it is appropriate to describe the evaluation process using JavaScript, which, after all, is our tool for describing processes.³ An evaluator

²The most important features that our evaluator leaves out are mechanisms for handling errors and supporting debugging. For a more extensive discussion of evaluators, see Friedman, Wand, and Haynes 1992, which gives an exposition of programming languages that proceeds via a sequence of evaluators written in the Scheme dialect of Lisp.

³Even so, there will remain important aspects of the evaluation process that are not elucidated by our evaluator. The most important of these are the detailed mechanisms by which functions call other functions and return values to their callers. We will address these issues in chapter 5, where we take a closer look at the evaluation

that is written in the same language that it evaluates is said to be *metacircular*.

The metacircular evaluator is essentially a JavaScript formulation of the environment model of evaluation described in section 3.2. Recall that the model has three basic parts:

- To evaluate an operator combination, evaluate the subexpressions and then apply the operator to the values of the subexpressions.
- To evaluate a function application combination, evaluate the function subexpression and the argument subexpressions, and then apply the value of the function subexpression to the values of the argument subexpressions.
- To apply a function to a set of arguments, evaluate the body of the function in a new environment. To construct this environment, extend the environment part of the function object by a frame in which the formal parameters of the function are bound to the arguments to which the function is applied.

These three rules describe the essence of the evaluation process, a basic cycle in which statements to be evaluated in environments are reduced to functions to be applied to arguments, which in turn are reduced to new statements to be evaluated in new environments, and so on, until we get down to symbols, whose values are looked up in the environment, and to operators, which are applied directly (see Figure 4.1).⁴ This evaluation cycle will be embodied by the interplay between the two critical functions in the evaluator, eval and apply, which are described in section 4.1.1 (see Figure 4.1).

The implementation of the evaluator will depend upon functions that define the *syntax* of the expressions to be evaluated. We will use data abstraction to make the evaluator independent

process by implementing the evaluator as a simple register machine.

⁴If we grant ourselves the ability to apply primitives, then what remains for us to implement in the evaluator? The job of the evaluator is not to specify the primitives of the language, but rather to provide the connective tissue—the means of combination and the means of abstraction—that binds a collection of primitives to form a language. Specifically:

- The evaluator enables us to deal with nested statements. For example, although simply applying primitives would suffice for evaluating the statement `1 + 6;`, it is not adequate for handling `1 + (2 * 3);`. As far as the primitive function `+` is concerned, its arguments must be numbers, and it would choke if we passed it the expression `2 * 3` as an argument. One important role of the evaluator is to choreograph function composition so that `2 * 3` is reduced to `6` before being passed as an argument to `+`.
- The evaluator allows us to use variables. For example, the primitive function for addition has no way to deal with expressions such as `x + 1`. We need an evaluator to keep track of variables and obtain their values before invoking the primitive functions.
- The evaluator allows us to define compound functions. This involves keeping track of function definitions, knowing how to use these definitions in evaluating expressions, and providing a mechanism that enables functions to accept arguments.
- The evaluator provides the special forms, which must be evaluated differently from function calls.

of the representation of the language. For example, rather than committing to a choice that an assignment is to be represented by a list beginning with the symbol assignment we use an abstract predicate `is_assignment` to test for an assignment, and we use abstract selectors `assignment_name` and `assignment_right_hand_side` to access the parts of an assignment. Implementation of expressions will be described in detail in section 4.1.2. There are also operations, described in section 4.1.3, that specify the representation of functions and environments. For example, `make_function_object` constructs compound functions, `lookup_name_value` accesses the values of variables, and `apply_builtin_function` applies a primitive function to a given list of arguments.

4.1.1 The Core of the Evaluator

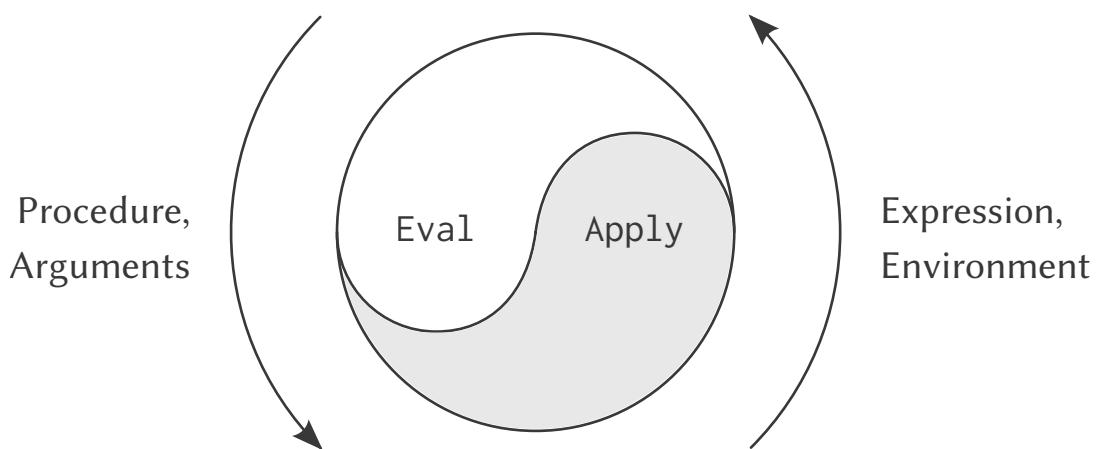


Figure 4.1: The eval–apply cycle exposes the essence of a computer language.

The evaluation process can be described as the interplay between two functions: `eval` and `apply`.

The function evaluate

The function `evaluate` takes as arguments a statement and an environment. It classifies the statement and directs its evaluation. The function `evaluate` is structured as a case analysis of the syntactic type of the expression to be evaluated. In order to keep the function general, we express the determination of the type of a statement abstractly, making no commitment to any particular representation for the various types of statements. Each type of statement has a predicate that tests for it and an abstract means for selecting its parts. This *abstract syntax* makes it easy to see how we can change the syntax of the language by using the same evaluator, but with a different collection of syntax functions.

Primitive expressions

- If the given statement is a self-evaluating expression, such as a number, evaluate returns the expression itself.
- The function evaluate must look up names in the environment to find their values.

Special forms

- An assignment to (or a declaration of) a name must recursively call evaluate to compute the new value to be associated with the variable. The environment must be modified to change (or create) the binding of the variable.
- A conditional expression requires special processing of its parts, so as to evaluate the consequent if the predicate is true, and otherwise to evaluate the alternative.
- A function definition expression must be transformed into an applicable function by packaging together the parameters and body specified by the function definition with the environment of the evaluation.
- A sequence of statements requires evaluating its component statements in the order in which they appear.
- A block requires evaluating its statements, while ensuring that declarations within the block remain local to the block.
- When evaluate encounters a **return** statement, the **return** expression is evaluated and marked as a return value.

Combinations

- For a function application, evaluate must recursively evaluate the operator part and the operands of the combination. The resulting function and arguments are passed to apply, which handles the actual function application.

Here is the definition of evaluate:

```
function evaluate(stmt, env) {
  return is_self_evaluating(stmt)
    ? stmt
    : is_name(stmt)
      ? lookup_name_value(name_of_name(stmt), env)
      : is_constant_declaration(stmt)
        ? eval_constant_declaration(stmt, env)
```

```

    : is_variable_declaraction(stmt)
      ? eval_variable_declaraction(stmt, env)
    : is_assignment(stmt)
      ? eval_assignment(stmt, env)
    : is_conditional_expression(stmt)
      ? eval_conditional_expression(stmt, env)
    : is_function_definition(stmt)
      ? eval_function_definition(stmt, env)
    : is_sequence(stmt)
      ? eval_sequence(sequence_actions(stmt), env)
    : is_block(stmt)
      ? eval_block(stmt, env)
    : is_return_statement(stmt)
      ? eval_return_statement(stmt, env)
    : is_application(stmt)
      ? apply(evaluate(operator(stmt), env),
              list_of_values(operands(stmt), env))
    : Error("Unknown statement type in evaluate",
           stmt);
}

```

For clarity, `evaluate` has been implemented as a case analysis using conditional expressions. The disadvantage of this is that our function handles only a few distinguishable types of statements, and no new ones can be defined without editing the definition of `evaluate`. In most interpreter implementations, dispatching on the type of an expression is done in a data-directed style. This allows a user to add new types of expressions that `evaluate` can distinguish, without modifying the definition of `evaluate` itself. (See exercise 4.2.)

Apply

The function `apply` takes two arguments, a function and a list of arguments to which the function should be applied. The function `apply` classifies functions into two kinds: It calls `apply_primitive_function` to apply primitives; it applies compound functions by sequentially evaluating the expressions that make up the body of the function. The environment for the evaluation of the body of a compound function is constructed by extending the base environment carried by the function to include a frame that binds the parameters of the function to the arguments to which the function is to be applied. Here is the definition of `apply`:

```

function apply(fun, args) {
  if (is_primitive_function(fun)) {
    return apply_primitive_function(fun, args);
  } else if (is_compound_function(fun)) {
    const result =
      evaluate(function_body(fun),
              extend_environment(

```

```

        function_parameters(fun),
        args,
        function_environment(fun)));
if (is_return_value(result)) {
    return return_value_content(result);
} else {
    return undefined;
}
} else {
    Error("Unknown function type in apply",
          fun);
}
}

```

In order to return a value, JavaScript functions need to evaluate a return statement. If a function terminates without return, the value undefined is returned. Thus, if the evaluation of the function body yields a return value, the corresponding return expression is returned, and otherwise the value undefined is returned.

Function arguments

When evaluate processes a function application, it uses list_of_values to produce the list of arguments to which the function is to be applied. The function list_of_values takes as an argument the operands of the combination. It evaluates each operand and returns a list of the corresponding values:⁵

```

function list_of_values(exp, env) {
    if (no_operands(exp)) {
        return null;
    } else {
        return pair(evaluate(first_operand(exp), env),
                    list_of_values(rest_operands(exp), env));
    }
}

```

⁵We could have simplified the is_application clause in evaluate by using map (and stipulating that operands returns a list) rather than writing an explicit list_of_values function. We chose not to use map here to emphasize the fact that the evaluator can be implemented without any use of higher-order functions (and thus could be written in a language that doesn't have higher-order functions), even though the language that it supports will include higher-order functions.

Conditionals

The function `eval_conditional_expression` evaluates the predicate part of an conditional expression in the given environment. If the result is true, the consequent is evaluated, otherwise the alternative:

```
function eval_conditional_expression(stmt, env) {
    return is_true(evaluate(cond_expr_pred(stmt),
                           env))
        ? evaluate(cond_expr_cons(stmt),
                   env)
        : evaluate(cond_expr_alt(stmt),
                   env);
}
```

The use of `is_true` in `eval_conditional_expression` highlights the issue of the connection between an implemented language and an implementation language. The predicate is evaluated in the language being implemented and thus yields a value in that language. The interpreter predicate `is_true` translates that value into a value that can be tested by the conditional expression in the implementation language: The metacircular representation of truth might not be the same as that of the underlying JavaScript.⁶

Function definitions

The evaluation of a function definition expression creates a function object that represents the function during the evaluation. The function object contains the parameters and the body of the function definition, as well as the environment with respect to which the function definition is evaluated. According to the environment model, this is the environment that needs to be extended, when the function gets applied to actual arguments.

```
function eval_function_definition(stmt, env) {
    return make_function(
        map(name_of_name,
            function_definition_parameters(stmt)),
        function_definition_body(stmt),
        env);
}
```

⁶In this case, the language being implemented and the implementation language are the same. Contemplation of the meaning of `is_true` here yields expansion of consciousness without the abuse of substance.

Sequences

The function eval_sequence is used by eval to evaluate a sequence of statements. Note that the evaluation of the first component of a sequence may yield a return value, in which case the rest of the statement is not evaluated.

```
function eval_sequence(stmts, env) {
  if (is_empty_statements(stmts)) {
    return undefined;
  } else if (is_last_statement(stmts)) {
    return evaluate(first_statement(stmts), env);
  } else {
    const first_stmt_value =
      evaluate(first_statement(stmts), env);
    if (is_return_value(first_stmt_value)) {
      return first_stmt_value;
    } else {
      return eval_sequence(
        rest_statements(stmts), env);
    }
  }
}
```

Blocks

The function eval_block is used by evaluate to evaluate block statements. The constants and variables declared in the block need to be local to the block. The evaluation of block statements evaluates the body of the block with respect to an environment that extends the current environment. Initially, there are no bindings in the innermost frame of this environment, but each declaration will add a new binding to it.

```
function eval_block(stmt, env) {
  return evaluate(block_body(stmt),
    extend_environment(null, null, env));
}
```

Return statements

The function `eval_return_statement` is used by `evaluate` to evaluate return statements. As seen in the evaluation of sequences, the result of evaluation of return statements needs to be identifiable so that the evaluation of function bodies can return immediately, even if there are statements after the return statement. For this purpose, the evaluation of a return statement wraps the result of evaluating the return expression in a return value object.

```
function eval_return_statement(stmt, env) {
    return make_return_value(
        evaluate(return_statement_expression(stmt),
            env));
}
```

Assignments and declarations

The following function handles assignments to variables. It calls `evaluate` to find the value to be assigned and transmits the variable and the resulting value to `assign_name_to_value` to be installed in the designated environment.

```
function eval_assignment(stmt, env) {
    const value = evaluate(assignment_value(stmt), env);
    set_variable_value(assignment_name(stmt), value, env);
    return value;
}
```

Declarations of constants and variables are handled in a similar manner. Section 4.1.3 explains how we distinguish variables and constants in the functions `declare_variable` and `declare_constant` and how we prevent assignment to constants.

```
function eval_variable_declaration(stmt, env) {
    declare_variable(variable_declaration_name(stmt),
        evaluate(variable_declaration_value(stmt), env),
        env);
}
function eval_constant_declaration(stmt, env) {
    declare_constant(constant_declaration_name(stmt),
        evaluate(constant_declaration_value(stmt), env),
        env);
}
```

Note that the returned value of constant and variable declaration is the value `undefined`, as prescribed by the ECMAScript standard (Ecma 1997).

Exercise 4.1

Notice that we cannot tell whether the metacircular evaluator evaluates operands from left to right or from right to left. Its evaluation order is inherited from the underlying JavaScript: If the arguments to `pair` in `list_of_values` are evaluated from left to right, then `list_of_values` will evaluate operands from left to right; and if the arguments to `pair` are evaluated from right to left, then `list_of_values` will evaluate operands from right to left. Write a version of `list_of_values` that evaluates operands from left to right regardless of the order of evaluation in the underlying JavaScript. Also write a version of `list_of_values` that evaluates operands from right to left.

4.1.2 Representing Statements and Expressions

The evaluator is reminiscent of the symbolic differentiation program discussed in section 2.3.2. Both programs operate on symbolic expressions. In both programs, the result of operating on a compound expression is determined by operating recursively on the pieces of the expression and combining the results in a way that depends on the type of the expression. In both programs we used data abstraction to decouple the general rules of operation from the details of how expressions are represented. In the differentiation program this meant that the same differentiation function could deal with algebraic expressions in prefix form, in infix form, or in some other form. For the evaluator, this means that the syntax of the language being evaluated is determined solely by the functions that classify and extract pieces of expressions.

Here is the specification of the syntax of our language:

- The self-evaluating items are numbers, strings and boolean values.

```
function is_self_evaluating(stmt) {
    return is_number(stmt) ||
           is_string(stmt) ||
           is_boolean(stmt);
}
```

- The function `is_name` tests whether the given statement is a name expression, and the function `name_of_name` accesses the JavaScript string that represents the name.

```
function is_name(stmt) {
    return is_tagged_list(stmt, "name");
}
function name_of_name(stmt) {
    return head(tail(stmt));
}
```

The function `is_name` is defined in terms of the function `is_tagged_list`, which identifies lists beginning with a designated string that we call *tag*:

```

|   function is_tagged_list(stmt, the_tag) {
|     return is_pair(stmt) && head(stmt) === the_tag;
| }
```

- Assignments have the form *name = value*:

```

|   function is_assignment(stmt) {
|     return is_tagged_list(stmt, "assignment");
| }
|   function assignment_name(stmt) {
|     return head(tail(head(tail(stmt)))));
| }
| function assignment_value(stmt) {
|   return head(tail(tail(stmt)));
| }
```

- Declarations have the form

const name = value;

or

let name = value;

or

```

function name(parameter1, ..., parametern) {
  body
}
```

Here, we treat the latter form (function declarations) as syntactic sugar⁷ for

const name = (parameter₁, ..., parameter_n) => { body; };

The corresponding syntax functions are the following:

```

|   function is_constant_declaration(stmt) {
|     return is_tagged_list(stmt, "constant_declaration");
| }
|   function constant_declaration_name(stmt) {
|     return head(tail(head(tail(stmt)))));
| }
| function constant_declaration_value(stmt) {
|   return head(tail(tail(stmt)));
| }
| function is_variable_declaration(stmt) {
|   return is_tagged_list(stmt, "variable_declaration");
| }
| function variable_declaration_name(stmt) {
```

⁷In actual JavaScript, there is a subtle difference between the two forms. The interpretation of function declaration statements involves reordering of sequence statements, a topic which we prefer to skip at this point.

```

    return head(tail(head(tail(stmt))));
}
function variable_declaration_value(stmt) {
    return head(tail(tail(stmt)));
}

```

- Function definitions are objects tagged with the string `function_definition`:

```

function is_function_definition(stmt) {
    return is_tagged_list(stmt, "function_definition");
}
function function_definition_parameters(stmt) {
    return head(tail(stmt));
}
function function_definition_body(stmt) {
    return head(tail(tail(stmt)));
}

```

- `return` statements are objects tagged with the string `"return_statement"`:

```

function is_return_statement(stmt) {
    return is_tagged_list(stmt, "return_statement");
}
function return_statement_expression(stmt) {
    return head(tail(stmt));
}

```

- Conditional expressions are tagged with `"conditional_expression"` and have a predicate, a consequent, and an alternative.

```

function is_conditional_expression(stmt) {
    return is_tagged_list(stmt,
                          "conditional_expression");
}
function cond_expr_pred(stmt) {
    return list_ref(stmt, 1);
}
function cond_expr_cons(stmt) {
    return list_ref(stmt, 2);
}
function cond_expr_alt(stmt) {
    return list_ref(stmt, 3);
}

```

- A sequence is a list of statements.

```

    function is_sequence(stmt) {
        return is_tagged_list(stmt, "sequence");
    }
    function sequence_actions(stmt) {
        return head(tail(stmt));
    }
    function is_empty_statements(stmts) {
        return is_null(stmts);
    }
    function is_last_statement(stmts) {
        return is_null(tail(stmts));
    }
    function first_statement(stmts) {
        return head(stmts);
    }
    function rest_statements(stmts) {
        return tail(stmts);
    }
}

```

- A block contains its body statement.

```

    function is_block(stmt) {
        return is_tagged_list(stmt, "block");
    }
    function block_body(stmt) {
        return head(tail(stmt));
    }
}

```

- A function application is an object tagged with the string "application". We provide access functions for the operator, the operands, and three functions for iterating through the operand list:

```

    function is_application(stmt) {
        return is_tagged_list(stmt, "application");
    }
    function operator(stmt) {
        return head(tail(stmt));
    }
    function operands(stmt) {
        return head(tail(tail(stmt)));
    }
    function no_operands(ops) {
        return is_null(ops);
    }
    function first_operand(ops) {
        return head(ops);
    }
    function rest_operands(ops) {
}

```

```
|     return tail(ops);  
| }
```

Exercise 4.2

Rewrite `evaluate` so that the dispatch is done in data-directed style. Compare this with the data-directed differentiation function of exercise 2.73. (You may use the head of a compound expression as the type of the expression, as is appropriate for the syntax implemented in this section.)

Exercise 4.3

Recall the definitions of the special forms `&&` and `||` from chapter 1:

- $expression_1 \&& expression_2$: The expression $expression_1$ is evaluated first. If it evaluates to `false`, `false` is returned; the expression $expression_2$ is not evaluated. If it evaluates to `true`, the value of $expression_2$ is returned.
- $expression_1 || expression_2$: The expression $expression_1$ is evaluated first. If it evaluates to `true`, `true` is returned; the expression $expression_2$ is not evaluated. If it evaluates to `false`, the value of $expression_2$ is returned.

Include `&&` and `||` expressions by defining appropriate syntax functions and evaluation functions `eval_and` and `eval_or`

4.1.3 Evaluator Data Structures

In addition to defining the external syntax of expressions, the evaluator implementation must also define the data structures that the evaluator manipulates internally, as part of the execution of a program, such as the representation of functions and environments and the representation of true and false.

Testing of predicates

To enter the consequent of a conditional, we expect the predicate to evaluate to the value `true`, and thus we define the evaluator function `is_true` as follows:

```
function is_true(x) {
    return x === true;
}
```

With the definition of the function `eval_conditional_statement` of section 4.1.1, this means that our evaluator evaluates the alternative statement for any predicate value other than `true`.

Representing functions

To handle primitives, we assume that we have available the following functions:

- `apply_primitive_function(fun, args)` applies the given primitive function to the argument values in the list `args` and returns the result of the application.
- `is_primitive_function(fun)` tests whether `fun` is a primitive function.

These mechanisms for handling primitives are further described in section 4.1.4.

Compound functions are constructed from parameters, function bodies, and environments using the constructor `make_function`:

```
function make_function(parameters, body, env) {
    return list("function",
                parameters, body, env);
}
function is_compound_function(f) {
    return is_tagged_list(f, "function");
}
function function_parameters(f) {
    return list_ref(f, 1);
}
function function_body(f) {
    return list_ref(f, 2);
```

```

}
function function_environment(f) {
    return list_ref(f, 3);
}

```

Representing return values

We saw in section 4.1.1 that the evaluation of sequences terminates with the first **return** statement encountered, and that the evaluation of function applications needs to return the value `undefined` if the evaluation of the function body does not encounter a **return** statement. In order to identify the evaluation of **return** statements, we introduce **return** values as evaluator data structures.

```

function make_return_value(content) {
    return list("return_value", content);
}
function is_return_value(value) {
    return is_tagged_list(value, "return_value");
}
function return_value_content(value) {
    return head(tail(value));
}

```

Operations on Environments

The evaluator needs operations for manipulating environments. As explained in section 3.2, an environment is a sequence of frames, where each frame is a table of bindings that associate names with their corresponding values. We use the following operations for manipulating environments:

- `lookup_name_value(name, env)` returns the value that is bound to the symbol *name* in the environment *env*, or signals an error if the name is unbound.
- `extend_environment(names, values, base-env)` returns a new environment, consisting of a new frame in which the symbols in the list *names* are bound to the corresponding elements in the list *values* (each tagged as *mutable*), where the enclosing environment is the environment *base-env*.
- `declare_constant(name, value, env)` adds to the first frame in the environment *env* a new binding that associates the name *name* with the value *value*, tagged as *immutable*.
- `declare_variable(name, value, env)` adds to the first frame in the environment *env* a new binding that associates the name *name* with the value *value*, tagged as *mutable*.

- `assign_name_to_value(name, value, env)` checks if the value associated to the name `name` is tagged as mutable, and if yes, changes its binding in the environment `env` so that the name is now bound to the value `value`, or signals an error if the name is unbound or its value tagged as immutable.

To implement these operations we represent an environment as a list of frames. The enclosing environment of an environment is the tail of the list. The empty environment is simply the empty list.

```
function enclosing_environment(env) {
    return tail(env);
}
function first_frame(env) {
    return head(env);
}
const the_empty_environment = null;
function is_empty_environment(env) {
    return is_null(env);
}
```

Each frame of an environment is represented as a pair of lists: a list of the variables bound in that frame and a list of the associated values.⁸

```
function make_frame(names, values) {
    return pair(names, values);
}
function frame_names(frame) {
    return head(frame);
}
function frame_values(frame) {
    return tail(frame);
}
```

The function `add_binding_to_frame` is used both by constant and variable declaration. A flag `mut` indicates that the declared name is mutable.

```
function isMutable(val) {
    return tail(val) === true;
}
function add_binding_to_frame(name, val, frame, mut) {
    set_head(frame, pair(name, head(frame)));
    set_tail(frame, pair(pair(val, mut), tail(frame)));
}
```

⁸Frames are not really a data abstraction in the following code: `set_variable_value` and `define_variable` use `set_head` to directly modify the values in a frame. The purpose of the frame functions is to make the environment-manipulation functions easy to read.

To extend an environment by a new frame that associates names with values, we make a frame consisting of the list of names and the list of values, and we adjoin this to the environment. We signal an error if the number of names does not match the number of values.

```
function extend_environment(names, vals, base_env) {
  if (length(names) === length(vals)) {
    return pair(make_frame(names,
                           map(x => pair(x, true), vals)),
               base_env);
  } else if (length(names) < length(vals)) {
    error("Too many arguments supplied: " +
          stringify(names) + ", " +
          stringify(vals));
  } else {
    error("Too few arguments supplied: " +
          stringify(names) + ", " +
          stringify(vals));
  }
}
```

The function `extend_environment` is used by `apply` in section 4.1.1 to bind the parameters of a function to its arguments. In order to allow for assignment to function parameters, as in function `make_withdraw_with_balance` of section 3.1.1, we choose to tag the values in `extend_environment` as mutable, using `map`.

To look up a name in an environment, we scan the list of name in the first frame. If we find the desired name, we return the corresponding element in the list of values. If we do not find the name in the current frame, we search the enclosing environment, and so on. If we reach the empty environment, we signal an ‘unbound name’ error.

```
function lookup_name_value(name, env) {
  function env_loop(env) {
    function scan(names, vals) {
      return is_null(names)
        ? env_loop(
            enclosing_environment(env))
        : name === head(names)
          ? head(head(vals))
          : scan(tail(names), tail(vals));
    }
    if (env === the_empty_environment) {
      Error("Unbound name", name);
    } else {
      const frame = first_frame(env);
      return scan(frame_names(frame),
                  frame_values(frame));
    }
}
```

```

    }
    return env_loop(env);
}

```

To assign a name to a new value in a specified environment, we scan for the name, just as in `lookup_name_value`, and change the corresponding value when we find it, provided it is tagged as mutable.

```

function set_variable_value(name, val, env) {
    function env_loop(env) {
        function scan(names, vals) {
            return is_null(name)
                ? env_loop(
                    enclosing_environment(env))
                : name === head(names)
                ? ( tail(head(vals))
                    ? set_head(head(vals), val)
                    : error("no assignment " +
                            "to constants allowed") )
                : scan(tail(names), tail(vals));
        }
        if (env === the_empty_environment) {
            Error("Unbound name in assignment",
                  name);
        } else {
            const frame = first_frame(env);
            return scan(frame_names(frame),
                      frame_values(frame));
        }
    }
    return env_loop(env);
}

```

To declare a constant/variable, we adjoin a binding for the name to the value in the first frame in the given environment, and tag the value as immutable/mutable.

```

function declare_constant(name, val, env) {
    add_binding_to_frame(
        name, val, first_frame(env), false);
}
function declare_variable(name, val, env) {
    add_binding_to_frame(
        name, val, first_frame(env), true);
}

```

The method described here is only one of many plausible ways to represent environments. Since we used data abstraction to isolate the rest of the evaluator from the detailed choice of representation, we could change the environment representation if we wanted to. (See exer-

cise 4.4.) In a production-quality JavaScript system, the speed of the evaluator’s environment operations—especially that of variable lookup—has a major impact on the performance of the system. The representation described here, although conceptually simple, is not efficient and would not ordinarily be used in a production system.⁹

Exercise 4.4

Instead of representing a frame as a pair of lists, we can represent a frame as a list of bindings, where each binding is a name-value pair. Rewrite the environment operations to use this alternative representation.

Exercise 4.5

JavaScript allows us to create new bindings for names by means of constant and variable declaration, but provides no way to get rid of bindings. Implement for the evaluator a ‘function’ `make_unbound` that removes the binding of a name given as ‘argument’ from the environment in which the application of the function is evaluated. This problem is not completely specified. For example, should we remove only the binding in the first frame of the environment? Complete the specification and justify any choices you make.

4.1.4 Running the Evaluator as a Program

Given the evaluator, we have in our hands a description (expressed in JavaScript) of the process by which JavaScript expressions are evaluated. One advantage of expressing the evaluator as a program is that we can run the program. This gives us, running within JavaScript, a working model of how JavaScript itself evaluates expressions. This can serve as a framework for experimenting with evaluation rules, as we shall do later in this chapter.

Our evaluator program reduces expressions ultimately to the application of primitive functions. Therefore, all that we need to run the evaluator is to create a mechanism that calls on the underlying JavaScript system to model the application of primitive functions.

There must be a binding for each primitive function name, so that when `evaluate` evaluates the operator of an application of a primitive, it will find an object to pass to `apply`. We thus set up a global environment that associates unique objects with the names of the primitive functions that can appear in the expressions we will be evaluating. The global environment also includes bindings for the symbols `undefined`, `Nan` and `Infinity`, so that they can be used as constants in expressions to be evaluated.

⁹The drawback of this representation (as well as the variant in exercise 4.4) is that the evaluator may have to search through many frames in order to find the binding for a given variable. (Such an approach is referred to as *deep binding*.) One way to avoid this inefficiency is to make use of a strategy called *lexical addressing*.

```

function setup_environment() {
  const initial_env = pair(make_frame(null, null),
                           the_empty_environment);
  for_each(x =>
    declare_constant(head(x),
                    make_primitive_function(head(tail(x))),
                    initial_env),
    primitive_functions);
  for_each(x =>
    declare_constant(head(x),
                    head(tail(x)),
                    initial_env),
    primitive_values);
  return initial_env;
}

```

It does not matter how we represent primitive functions, so long as apply can identify and apply them using the functions `is_primitive_function` and `apply_primitive_function`. We have chosen to represent a primitive function as a list beginning with the string "primitive" and containing a function in the underlying JavaScript that implements that primitive.

```

function make_primitive_function(impl) {
  return list("primitive", impl);
}
function is_primitive_function(fun) {
  return is_tagged_list(fun, "primitive");
}
function primitiveImplementation(fun) {
  return list_ref(fun, 1);
}

```

The function `setup_environment` will get the primitive names and implementation functions from a list:¹⁰

```

const primitive_functions = list(
  list("display",           display          ),
  list("error",             error            ),
  list("+",                 (x, y) => x + y  ),
  list("-",                 (x, y) => x - y  ),
  list("*",                 (x, y) => x * y  ),
  list("/",                 (x, y) => x / y  ),
  list("%",                 (x, y) => x % y  ),
  list("===",                (x, y) => x === y),
)

```

¹⁰Any function defined in the underlying JavaScript can be used as a primitive for the metacircular evaluator. The name of a primitive installed in the evaluator need not be the same as the name of its implementation in the underlying JavaScript; the names are the same here because the metacircular evaluator implements JavaScript itself. Thus, for example, we could put `list("first", head)` or `list("square", x => x * x)` in the list of `primitive_functions`.

```

list("!=",
  list("<",
    list("<=",
      list(">",
        list(">=",
          list("!",
            x => ! x
          );
        )
      )
    )
  )
);

```

Similar to primitive functions, we define primitive values that are installed in the global environment by the function `setup_environment`.

```

const primitive_values = list(list("undefined", undefined),
  list("NaN", NaN),
  list("Infinity", Infinity),
  list("math_PI", math_PI)
);

```

To apply a primitive function, we simply apply the implementation function to the arguments, using the underlying JavaScript system:¹¹

```

function apply_primitive_function(fun, argument_list) {
  return apply_in_underlying_javascript(
    primitive_implementation(fun),
    argument_list);
}

```

In JavaScript, `return` statements are only allowed within function bodies. Any evaluation of such statements outside of function bodies should lead to an error, a service provided by the function `eval_toplevel`.

```

function eval_toplevel(stmt) {
  const value = evaluate(stmt, the_global_environment);
  if (is_return_value(value)) {
    error("return not allowed " +
      "outside of function definitions");
  } else {
    return value;
}

```

¹¹JavaScript's `apply` method of function objects expects arguments in an array. Thus, the `argument_list` is transformed into an array using a `while` loop:

```

function apply_in_underlying_javascript(prim, argument_list) {
  const argument_array = [];
  let i = 0;
  while (!is_null(argument_list)) {
    argument_array[i] = head(argument_list);
    i = i + 1;
    argument_list = tail(argument_list);
  }
  return prim.apply(prim, argument_array);
}

```

We have made use of `apply_in_underlying_javascript` in to define the function `apply` in section 2.4.3.

```

    }
}
```

For convenience in running the metacircular evaluator, we provide a *read-eval-print loop*. It prints a *prompt*, reads an input expression from a pop-up window, evaluates this expression in the global environment, and prints the result on the next pop-up window.

```

function read_eval_print_loop(history) {
  const prog = prompt("History:" + history +
                      "\n\n" + "Enter next: ");
  if (prog === "") {
    prompt("session has ended");
  } else {
    const res = parse_and_eval(prog);
    read_eval_print_loop(history + "\n" +
                          stringify(prog) + " ==> " +
                          stringify(user_print(res)));
  }
}
```

The function `parse_and_eval` transforms a statement string into a tagged-object representation of the statement according to the description in section 4.1.2, a process called *parsing* and accomplished by the primitive function `parse`. After that, it applies the function `eval_toplevel` to the tagged-object representation.

```

function parse_and_eval(str) {
  return eval_toplevel(parse(str));
}
```

We use a special printing function `user_print`, to avoid printing the environment part of a compound function, which may be a very long list (or may even contain cycles).

```

function user_print(object) {
  return is_compound_function(object)
    ? "function" +
      stringify(function_parameters(object)) +
      stringify(function_body(object)) +
      "<environment>"
    : object;
}
```

Now all we need to do to run the evaluator is to initialize the global environment and start the driver loop. Here is a sample interaction:

```

const the_global_environment = setup_environment();
read_eval_print_loop("");
```

Exercise 4.6

The global environment includes a binding for the symbol `undefined`. This way of treating `undefined` is consistent with JavaScript's definition; it is a variable and not a keyword in the language as are `true` and `false`. A disadvantage of this treatment of `undefined` is that it could be redefined as in

```
const undefined = "defined";
```

Some internet browsers such as Firefox silently prevent the redefinition of `undefined` such that the line above has no effect. Modify the interpreter such that `undefined` can be neither assigned nor re-defined using variable statements. In this implementation, the variable `undefined` must always refer to JavaScript's value `undefined`.

The Firefox browser (Version 14) allowed the re-definition of the variable `undefined` in function definitions, such that

```
function f(undefined) {
    return undefined + 1;
}
f(2);
```

produces the result 3. Modify your interpreter such that constant declarations cannot change `undefined`, but function definitions can.

Exercise 4.7

Eva Lu Ator and Louis Reasoner are each experimenting with the metacircular evaluator. Eva types in the definition of `map`, and runs some test programs that use it. They work fine. Louis, in contrast, has installed the system version of `map` as a primitive for the metacircular evaluator. When he tries it, things go terribly wrong. Explain why Louis's `map` fails even though Eva's works.

4.1.5 Data as Programs

In thinking about a JavaScript program that evaluates JavaScript expressions, an analogy might be helpful. One operational view of the meaning of a program is that a program is a description of an abstract (perhaps infinitely large) machine. For example, consider the familiar program to compute factorials:

```
function factorial(n) {
    return n === 1
        ? 1
        : factorial(n - 1) * n;
}
```

We may regard this program as the description of a machine containing parts that decrement, multiply, and test for equality, together with a two-position switch and another factorial ma-

chine. (The factorial machine is infinite because it contains another factorial machine within it.) Figure 4.2 is a flow diagram for the factorial machine, showing how the parts are wired together.

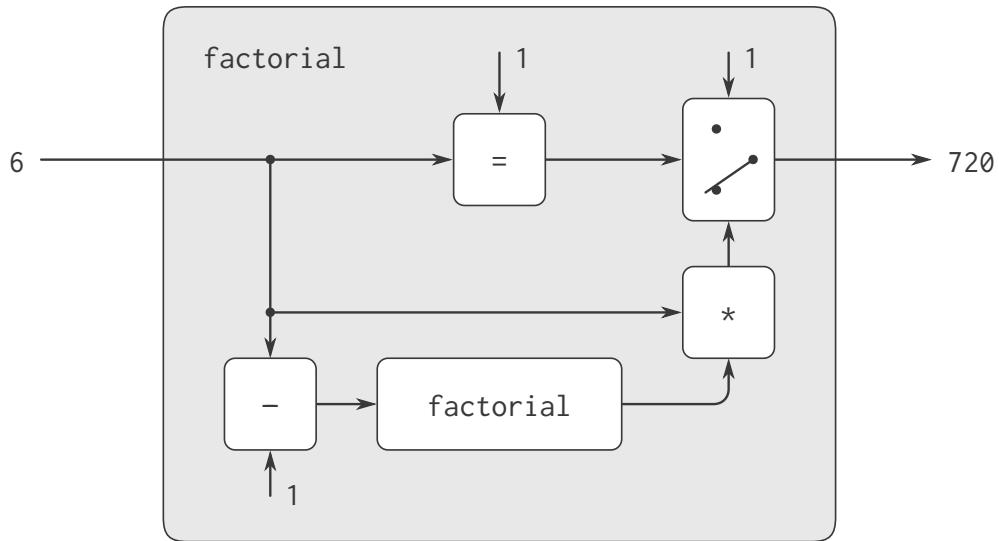


Figure 4.2: The factorial program, viewed as an abstract machine.

In a similar way, we can regard the evaluator as a very special machine that takes as input a description of a machine. Given this input, the evaluator configures itself to emulate the machine described. For example, if we feed our evaluator the definition of factorial, as shown in Figure 4.3, the evaluator will be able to compute factorials.

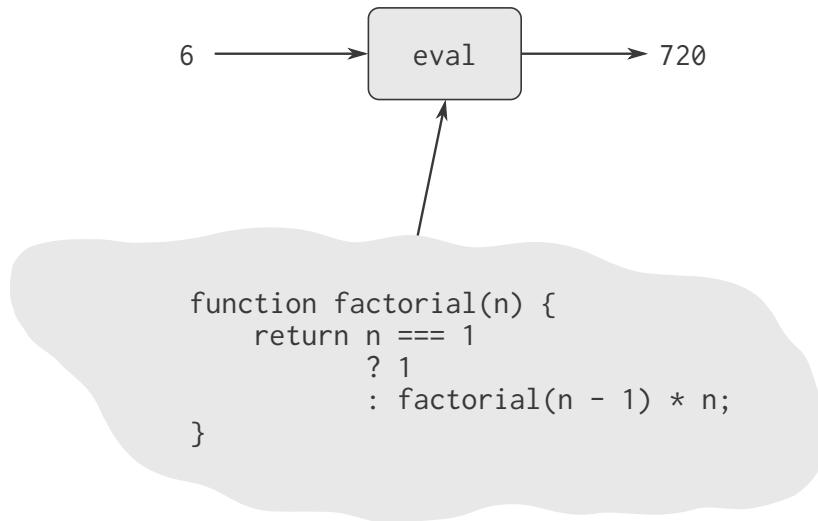


Figure 4.3: The evaluator emulating a factorial machine.

From this perspective, our evaluator is seen to be a *universal machine*. It mimics other machines when these are described as JavaScript programs.¹²

¹²The fact that the machines are described in JavaScript is inessential. If we give our evaluator a JavaScrip

This is striking. Try to imagine an analogous evaluator for electrical circuits. This would be a circuit that takes as input a signal encoding the plans for some other circuit, such as a filter. Given this input, the circuit evaluator would then behave like a filter with the same description. Such a universal electrical circuit is almost unimaginably complex. It is remarkable that the program evaluator is a rather simple program.¹³

Another striking aspect of the evaluator is that it acts as a bridge between the data objects that are manipulated by our programming language and the programming language itself. Imagine that the evaluator program (implemented in JavaScript) is running, and that a user is typing expressions to the evaluator and observing the results. From the perspective of the user, an input expression such as $x * x$ is an expression in the programming language, which the evaluator should execute. From the perspective of the evaluator, however, the expression is simply a string or—after parsing—a tagged-object representation that is to be manipulated according to a well-defined set of rules.

That the user’s programs are the evaluator’s data need not be a source of confusion. In fact, it is sometimes convenient to ignore this distinction, and to give the user the ability to explicitly evaluate a string as a JavaScript expression, using JavaScript’s primitive function `eval` that takes as argument a string. It parses the string and—provided that it syntactically correct—evaluates the resulting representation in the environment in which `eval` is applied.¹⁴

Exercise 4.8

program that behaves as an evaluator for some other language, say C, the JavaScript evaluator will emulate the C evaluator, which in turn can emulate any machine described as a C program. Similarly, writing a JavaScript evaluator in C produces a C program that can execute any JavaScript program. The deep idea here is that any evaluator can emulate any other. Thus, the notion of ‘what can in principle be computed’ (ignoring practicalities of time and memory required) is independent of the language or the computer, and instead reflects an underlying notion of *computability*. This was first demonstrated in a clear way by Alan M. Turing (1912–1954), whose 1936 paper laid the foundations for theoretical computer science. In the paper, Turing presented a simple computational model—now known as a *Turing machine*—and argued that any ‘effective process’ can be formulated as a program for such a machine. (This argument is known as the *Church-Turing thesis*.) Turing then implemented a universal machine, i.e., a Turing machine that behaves as an evaluator for Turing-machine programs. He used this framework to demonstrate that there are well-posed problems that cannot be computed by Turing machines (see exercise 4.8), and so by implication cannot be formulated as ‘effective processes.’ Turing went on to make fundamental contributions to practical computer science as well. For example, he invented the idea of structuring programs using general-purpose subroutines. See Hodges 1983 for a biography of Turing.

¹³Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple function, can emulate programs that are more complex than the evaluator itself. The existence of a universal evaluator machine is a deep and wonderful property of computation. *Recursion theory*, a branch of mathematical logic, is concerned with logical limits of computation. Douglas Hofstadter’s beautiful book *Gödel, Escher, Bach* (1979) explores some of these ideas.

¹⁴Warning: This `eval` primitive is not identical to the `evaluate` function we implemented in section 4.1.1, because it uses *actual* JavaScript environments rather than the sample environment structures we built in section 4.1.3. These actual environments cannot be manipulated by the user as ordinary lists; they must be accessed via `eval` or other special operations. Similarly, the `apply` primitive we saw in section 2.4.3 is not identical to the metacircular `apply`, because it uses actual JavaScript functions rather than the function objects we constructed in sections 4.1.3 and 4.1.4.

Given a one-argument function p and an object a , p is said to ‘halt’ on a if evaluating the expression $p(a)$ returns a value (as opposed to terminating with an error message or running forever). Show that it is impossible to write a function halts that correctly determines whether p halts on a for any function p and object a . Use the following reasoning: If you had such a function halts , you could implement the following program:

```
function run_forever() {
    return run_forever();
}

function try(p) {
    return halts(p, p)
        ? run_forever();
        : "halted";
}
```

Now consider evaluating the expression **try(try)** and show that any possible outcome (either halting or running forever) violates the intended behavior of halts .¹⁵

4.1.6 Internal Declarations

Our environment model of evaluation and our metacircular evaluator execute declarations in sequence, extending the environment frame one declaration at a time. This is particularly convenient for interactive program development, in which the programmer needs to freely mix the application of functions with the declaration of new functions. However, if we think carefully about the internal declarations used to implement block structure (introduced in section 1.1.8), we will find that name-by-name extension of the environment may not be the best way to declare local names.

Consider a function with internal declarations, such as

```
function f(x) {
    function is_even(n) {
        return n === 0
            ? true
            : is_odd(n - 1);
    }
    function is_odd(n) {
        return n === 0
            ? false
            : is_even(n - 1);
    }
    // rest of body of f
```

¹⁵Although we stipulated that halts is given a function object, notice that this reasoning still applies even if halts can gain access to the function’s text and its environment. This is Turing’s celebrated *Halting Theorem*, which gave the first clear example of a *non-computable* problem, i.e., a well-posed task that cannot be carried out as a computational function.

```
}
```

Our intention here is that the name `is_odd` in the body of the function `is_even` should refer to the function `is_odd` that is declared after `is_even`. The scope of the name `is_odd` is the entire body of `f`, not just the portion of the body of `f` starting at the point where the declaration of `is_odd` occurs. Indeed, when we consider that `is_odd` is itself defined in terms of `is_even`—so that `is_even` and `is_odd` are mutually recursive functions—we see that the only satisfactory interpretation of the two declarations is to regard them as if the names `is_even` and `is_odd` were being added to the environment simultaneously. More generally, in block structure, the scope of a local name is the entire function body in which the declaration is evaluated.

As it happens, our interpreter will evaluate calls to `f` correctly, but for an ‘accidental’ reason: Since the declarations of the internal functions come first, no calls to these functions will be evaluated until all of them have been declared. Hence, `is_odd` will have been declared by the time `is_even` is executed. In fact, our sequential evaluation mechanism will give the same result as a mechanism that directly implements simultaneous declaration for any function in which the internal declarations come first in a body and evaluation of the value expressions for the declared names doesn’t actually use any of the declared names. (For an example of a function that doesn’t obey these restrictions, so that sequential declaration isn’t equivalent to simultaneous declaration, see exercise 4.12.)¹⁶

There is, however, a simple way to treat declarations so that internally declared names have truly simultaneous scope—just create all local names that will be in the current environment before evaluating any of the value expressions. One way to do this is by a syntax transformation on function definition expressions.¹⁷ Before evaluating the body of a function definition expression, we ‘scan out’ and eliminate all the internal declarations in the body. The internally declared names will be created with a function definition and then set to their values by assignment. In the following, we shall focus on variable declarations using `let`; constant declarations using `const` and `function` can be handled similarly. For example, the function definition

```
(vars) => {
  let u = e1;
  let v = e2;
  statement
}
```

would be transformed into

¹⁶Wanting programs to not depend on this evaluation mechanism is the reason for the ‘management is not responsible’ remark in footnote 17 of chapter 1. The designers of JavaScript chose to resolve this issue by moving all internal function declarations to the beginning of the function body, and thus the discussion might seem moot. However, this mechanism is only applied to function declarations and not to `const` declarations.

¹⁷We can view function declaration statements as a combination of constant declaration statements and function definition expressions, as explained in section 1.3.2, and thus the same technique applies to function declaration statements.

```
( vars ) => {
    return ( (u, v) => {
        u = e1;
        v = e2;
        statement
    })("unassigned", "unassigned");
}
```

where “*unassigned*” is a special symbol that causes looking up a name to signal an error if an attempt is made to use the value of the not-yet-assigned name.

An alternative strategy for scanning out internal declarations is shown in exercise 4.11. Unlike the transformation shown above, this enforces the restriction that the declared names’ values can be evaluated without using any of the names’ values.

Exercise 4.9

In this exercise we implement the method just described for interpreting internal definitions.

- a. Change `lookup_name_value` (section 4.1.3) to signal an error if the value it finds is the string “*unassigned*”.
- b. Write a function `scan_out_let` that takes a function body and returns an equivalent one that has no internal definitions, by making the transformation described above.
- c. Install `scan_out_let` in the interpreter, either in `make_function` or in `function_body` (see section 4.1.3). Which place is better? Why?

Exercise 4.10

Draw diagrams of the environment in effect when evaluating the *statement* in the function in the text, comparing how this will be structured when declarations are interpreted sequentially with how it will be structured if declarations are scanned out as described. Why is there an extra frame in the transformed program? Explain why this difference in environment structure can never make a difference in the behavior of a correct program. Design a way to make the interpreter implement the ‘simultaneous’ scope rule for internal declarations without constructing the extra frame.

Exercise 4.11

Consider an alternative strategy for scanning out declarations that translates the example in the text to

```
( vars ) => {
    return ( (u, v) => {
        return ( (a, b) => {
            u = a;
            v = b;
            statement
        })(e1, e2);
    })("unassigned", "unassigned");
}
```

Here `a` and `b` are meant to represent new variable names, created by the interpreter, that do not appear in the user's program. Consider the `solve` function from section 3.5.4:

```
function solve(f, y0, dt) {
    const y = integral( () => dy, y0, dt);
    const dy = stream_map(f, y);
    return y;
}
```

Will this function work if internal definitions are scanned out as shown in this exercise? What if they are scanned out as shown in the text? Explain.

Exercise 4.12

Ben Bitdiddle, Alyssa P. Hacker, and Eva Lu Ator are arguing about the desired result of evaluating the expression

```
let a = 1;
function f(x) {
    let b = a + x;
    let a = 5;
    return a + b;
}
f(10);
```

Ben asserts that the result should be obtained using the sequential rule for `let`: `b` is declared to be 11, then `a` is declared to be 5, so the result is 16. Alyssa objects that mutual recursion requires the simultaneous scope rule for internal function declarations, and that it is unreasonable to treat function names differently from other names. Thus, she argues for the mechanism implemented in exercise 4.9. This would lead to `a` being unassigned at the time that the value for `b` is to be computed. Hence, in Alyssa's view the function should produce an error. Eva has a third opinion. She says that if the declarations of `a` and `b` are truly meant to be simultaneous, then the value 5 for `a` should be used in evaluating `b`. Hence, in Eva's view `a` should be 5, `b` should be 15, and the result should be 20. Which (if any) of these viewpoints do you support?

Can you devise a way to implement internal declarations so that they behave as Eva prefers?¹⁸

Exercise 4.13

For recursion, we currently make use of the fact that the scope of a constant declaration is the surrounding block. An occurrence of the function name in its body can refer to the function, because it lies in the scope of the **const** that declares the name. Louis Reasoner thinks that there ought to be a way to specify recursive functions without using **const**, **let** or **function**. Amazingly, Louis's intuition is correct. It is indeed possible to specify recursive functions without using **const** or **let**, **function**, although the method for accomplishing this is much more subtle than Louis imagined. The following expression computes 10 factorial by applying a recursive factorial function.¹⁹

```
(n => (fact => fact(fact, n))
  ( (ft, k) => k === 1
    ? 1
    : k * ft(ft, k - 1)
  )
)
(10);
```

- Check (by evaluating the expression) that this really does compute factorials. Devise an analogous expression for computing Fibonacci numbers.
- Consider the following function, which includes mutually recursive internal definitions:

```
function f(x) {
  function is_even(n) {
    return n === 0
      ? true
      : is_odd(n - 1);
  }
  function is_odd(n) {
    return n === 0
      ? false
      : is_even(n - 1);
  }
}
```

¹⁸The designers of JavaScript support Alyssa on the following grounds: Eva is in principle correct—the definitions should be regarded as simultaneous. But it seems difficult to implement a general, efficient mechanism that does what Eva requires. In the absence of such a mechanism, it is better to generate an error in the difficult cases of simultaneous definitions (Alyssa's notion) than to produce an incorrect answer (as Ben would have it).

¹⁹This example illustrates a programming trick for formulating recursive functions without using **const**, **let** or **function**. The most general trick of this sort is the *Y operator*, which can be used to give a ‘pure λ -calculus’ implementation of recursion. (See Stoy 1977 for details on the lambda calculus, and Gabriel 1988 for an exposition of the *Y* operator in Scheme.)

```

    return is_even(x);
}

```

Fill in the missing expressions to complete an alternative definition of `f`, which uses neither `const` nor `let` nor internal function declarations:

```

function f(x) {
  return (
    (is_even, is_odd) =>
      is_even(is_even, is_odd, x)
  )
  ( (ev, od, n) =>
    n === 0 ? true : od(??, ??, ??),
    (ev, od, n) =>
      n === 0 ? false : ev(??, ??, ??)
  );
}

```

4.1.7 Separating Syntactic Analysis from Execution

The evaluator implemented above is simple, but it is very inefficient, because the syntactic analysis of expressions is interleaved with their execution. Thus if a program is executed many times, its syntax is analyzed many times. Consider, for example, evaluating `factorial(4)` using the following definition of `factorial`:

```

function factorial(n) {
  return n === 1
  ?
  factorial(n - 1) * n;
}

```

Each time `factorial` is called, the evaluator must determine that the body is a conditional expression and extract the predicate. Only then can it evaluate the predicate and dispatch on its value. Each time it evaluates the expression `factorial(n - 1) * n`, or the subexpressions `factorial(n - 1)` and `n - 1`, the evaluator must perform the case analysis in `evaluate` to determine that the expression is an application, and must extract its operator and operands. This analysis is expensive. Performing it repeatedly is wasteful.

We can transform the evaluator to be significantly more efficient by arranging things so that syntactic analysis is performed only once.²⁰ We split `evaluate`, which takes an expression and an environment, into two parts. The function `analyze` takes only the expression. It performs the syntactic analysis and returns a new function, the *execution function*, that encapsulates

²⁰This technique is an integral part of the compilation process, which we shall discuss in chapter 5. Jonathan Rees wrote a Scheme interpreter like this in about 1982 for the T project (Rees and Adams 1982). Marc Feeley (1986) (see also Feeley and Lapalme 1987) independently invented this technique in his master's thesis.

the work to be done in executing the analyzed expression. The execution function takes an environment as its argument and completes the evaluation. This saves work because `analyze` will be called only once on an expression, while the execution function may be called many times.

With the separation into analysis and execution, `evaluate` now becomes

```
function evaluate(exp, env) {
    return (analyze(exp))(env);
}
```

The result of calling `analyze` is the execution function to be applied to the environment. The `analyze` function is the same case analysis as performed by the original `eval` of section 4.1.1, except that the functions to which we dispatch perform only analysis, not full evaluation:

```
function analyze(stmt) {
    return is_self_evaluating(stmt)
        ? analyze_self_evaluating(stmt)
        : is_name(stmt)
        ? analyze_name(stmt)
        : is_constant_declaration(stmt)
        ? analyze_constant_declaration(stmt)
        : is_variable_declaration(stmt)
        ? analyze_variable_declaration(stmt)
        : is_assignment(stmt)
        ? analyze_assignment(stmt)
        : is_conditional_statement(stmt)
        ? analyze_conditional_statement(stmt)
        : is_function_definition(stmt)
        ? analyze_function_definition(stmt)
        : is_sequence(stmt)
        ? analyze_sequence(sequence_actions(stmt))
        : is_block(stmt)
        ? analyze_block(block_body(stmt))
        : is_return_statement(stmt)
        ? analyze_return_statement(stmt)
        : is_application(stmt)
        ? analyze_application(stmt)
        : Error("Unknown statement type in analyze",
               stmt);
}
```

Here is the simplest syntactic analysis function, which handles self-evaluating expressions. It returns an execution function that ignores its environment argument and just returns the expression:

```
function analyze_self_evaluating(stmt) {
    return env => stmt;
}
```

Looking up the value of a name must still be done in the execution phase, since this depends upon knowing the environment.²¹

```
function analyze_name(stmt) {
    return env => lookup_name_value(
        name_of_name(stmt, env));
}
```

The function `analyze_assignment` also must defer actually setting the variable until the execution, when the environment has been supplied. However, the fact that the `assignment_value` expression can be analyzed (recursively) during analysis is a major gain in efficiency, because the `assignment_value` expression will now be analyzed only once. The same holds true for constant and variable declarations.

```
function analyze_assignment(stmt) {
    const variable = assignment_variable(stmt);
    const vfun = analyze(assignment_value(stmt));
    return env => {
        set_variable_value(variable,
            vfun(env), env);
        return "ok";
    };
}
function analyze_variable_declaration(stmt) {
    const name =
        variable_declaration_name(stmt);
    const vfun =
        variable_declaration_value(stmt);
    return env => {
        declare_variable(name,
            vfun(env),
            env);
        return "ok";
    };
}
function analyze_constant_declaration(stmt) {
    const name =
        constant_declaration_name(stmt);
    const vfun =
        constant_declaration_value(stmt);
    return env => {
        declare_constant(name,
            vfun(env),
            env);
        return "ok";
    };
}
```

²¹There is, however, an important part of the variable search that *can* be done as part of the syntactic analysis.

For conditional statements, we extract and analyze the predicate, consequent, and alternative at analysis time.

```
function analyze_conditional_statement(stmt) {
  const pfun =
    analyze(cond_stmt_pred(stmt));
  const cfun =
    analyze(cond_stmt_cons(stmt));
  const afun =
    analyze(cond_stmt_alt(stmt));
  return env => is_true(pfun(env))
    ? cfun(env)
    : afun(env);
}
```

Analyzing a lambda expression also achieves a major gain in efficiency: We analyze the lambda body only once, even though functions resulting from evaluation of the lambda may be applied many times.

```
function analyze_function_definition(stmt) {
  const vars =
    function_definition_parameters(stmt);
  const bfun =
    analyze(function_definition_body(stmt));
  return env =>
    make_function(vars, bfun, env);
}
```

Analysis of a sequence of statements is more involved.²² Each statement in the sequence is analyzed, yielding an execution function. These execution functions are combined to produce an execution function that takes an environment as argument and sequentially calls each individual execution function with the environment as argument.

```
function analyze_sequence(stmts) {
  function sequentially(fun1, fun2) {
    return env => {
      fun1(env);
      fun2(env);
    };
  }
  function loop(first_fun, rest_funs) {
    return is_null(rest_funs)
      ? first_fun
      : loop(sequentially(first_fun,
        head(rest_funs)),
        tail(rest_funs));
  }
  const funs = map(analyze, stmts);
```

²²See exercise 4.15 for some insight into the processing of sequences.

```

return is_null(funs)
    ? env => undefined
    : loop(head(funs), tail(funs));
}

```

To analyze an application, we analyze the operator and operands and construct an execution function that calls the operator execution function (to obtain the actual function to be applied) and the operand execution functions (to obtain the actual arguments). We then pass these to execute_application, which is the analog of apply in section 4.1.1. The function execute_application differs from apply in that the function body for a compound function has already been analyzed, so there is no need to do further analysis. Instead, we just call the execution function for the body on the extended environment.

```

function analyze_application(stmt) {
    const ffun = analyze(operator(stmt));
    const afuns = map(analyze, operands(stmt));
    return env =>
        execute_application(ffun(env),
            map(afun => afun(env), afuns));
}
function execute_application(fun, args) {
    return is_primitive_function(fun)
        ? apply_primitive_function(fun, args)
        : is_compound_function(fun)
            ? (function_body(fun))
                (extend_environment(
                    function_parameters(fun),
                    args,
                    function_environment(fun)))
            : Error("unknown function type in " +
                "execute_application",
                fun);
}

```

Our new evaluator uses the same data structures, syntax functions, and run-time support functions as in sections 4.1.2, 4.1.3, and 4.1.4.

Exercise 4.14

Extend the evaluator in this section to support conditional expressions.

Exercise 4.15

Alyssa P. Hacker doesn't understand why analyze_sequence needs to be so complicated. All the other analysis functions are straightforward transformations of the corresponding evalu-

ation functions (or eval clauses) in section 4.1.1. She expected `analyze_sequence` to look like this:

```
function analyze_sequence(stmts) {
  function execute_sequence(fun, env) {
    if (is_null(tail(fun))) {
      return head(fun)(env);
    } else {
      head(fun)(env);
      execute_sequence(tail(fun),
                      env);
    }
  }
  const funs = map(analyze, stmts);
  return is_null(funs)
    ? env => undefined
    : env => execute_sequence(funs,
                               env);
}
```

Eva Lu Ator explains to Alyssa that the version in the text does more of the work of evaluating a sequence at analysis time. Alyssa's `sequence_execution` function, rather than having the calls to the individual execution functions built in, loops through the functions in order to call them: In effect, although the individual expressions in the sequence have been analyzed, the sequence itself has not been. Compare the two versions of `sequence_execution`. For example, consider the common case (typical of function bodies) where the sequence has just one expression. What work will the execution function produced by Alyssa's program do? What about the execution function produced by the program in the text above? How do the two versions compare for a sequence with two expressions?

Exercise 4.16

Design and carry out some experiments to compare the speed of the original metacircular evaluator with the version in this section. Use your results to estimate the fraction of time that is spent in analysis versus execution for various functions.

4.2 Lazy Evaluation

Now that we have an evaluator expressed as a JavaScript program, we can experiment with alternative choices in language design simply by modifying the evaluator. Indeed, new languages are often invented by first writing an evaluator that embeds the new language within an existing high-level language. For example, if we wish to discuss some aspect of a proposed modification to JavaScript with another member of the JavaScript community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications. Not only does the high-level implementation base make it easier to test and debug the evaluator; in addition, the embedding enables the designer to *snarf*²³ features from the underlying language, just as our embedded JavaScript evaluator uses primitives and control structure from the underlying JavaScript. Only later (if ever) need the designer go to the trouble of building a complete implementation in a low-level language or in hardware. In this section and the next we explore some variations on JavaScript that provide significant additional expressive power.

4.2.1 Normal Order and Applicative Order

In section 1.1, where we began our discussion of models of evaluation, we noted that JavaScript is an *applicative-order* language, namely, that all the arguments to JavaScript functions are evaluated when the function is applied. In contrast, *normal-order* languages delay evaluation of function arguments until the actual argument values are needed. Delaying evaluation of function arguments until the last possible moment (e.g., until they are required by a primitive operation) is called *lazy evaluation*.²⁴ Consider the function

```
function try_me(a, b) {
    return a === 0 ? 1 : b;
}
```

Evaluating `try_me(0, head(null))`; generates an error in JavaScript. With lazy evaluation, there would be no error. Evaluating the expression would return 1, because the argument `head(null)` would never be evaluated.

An example that exploits lazy evaluation is the definition of a function `unless` that can be used in expressions such as

```
unless(xs === null,
```

²³Snarf: ‘To grab, especially a large document or file for the purpose of using it either with or without the owner’s permission.’ Snarf down: ‘To snarf, sometimes with the connotation of absorbing, processing, or understanding.’ (These definitions were snarfed from Steele et al. 1983. See also Raymond 1993.)

²⁴The difference between the ‘lazy’ terminology and the ‘normal-order’ terminology is somewhat fuzzy. Generally, ‘lazy’ refers to the mechanisms of particular evaluators, while ‘normal-order’ refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably.

```
head(xs),
display("error: xs should not be null"));
```

This won't work in an applicative-order language because both the usual value and the exceptional value will be evaluated before `unless` is called (compare exercise 1.6). An advantage of lazy evaluation is that some functions, such as `unless`, can do useful computation even if evaluation of some of their arguments would produce errors or would not terminate.

If the body of a function is entered before an argument has been evaluated we say that the function is *non-strict* in that argument. If the argument is evaluated before the body of the function is entered we say that the function is *strict* in that argument.²⁵

In a purely applicative-order language, all functions are strict in each argument. In a purely normal-order language, all compound functions are non-strict in each argument, and primitive functions may be either strict or non-strict. There are also languages (see exercise 4.23) that give programmers detailed control over the strictness of the functions they define.

A striking example of a function that can usefully be made non-strict is `pair` (or, in general, almost any constructor for data structures). One can do useful computation, combining elements to form data structures and operating on the resulting data structures, even if the values of the elements are not known. It makes perfect sense, for instance, to compute the length of a list without knowing the values of the individual elements in the list. We will exploit this idea in section 4.2.3 to implement the streams of chapter 3 as lists formed of non-strict `pair` pairs.

Exercise 4.17

Suppose that (in ordinary applicative-order JavaScript) we define `unless` as shown above and then define `factorial` in terms of `unless` as

```
function factorial(n) {
  return unless(n === 1,
               n * factorial(n - 1),
               1);
}
```

What happens if we attempt to evaluate `factorial(5)`? Will our definitions work in a normal-order language?

Exercise 4.18

²⁵The ‘strict’ versus ‘non-strict’ terminology means essentially the same thing as ‘applicative-order’ versus ‘normal-order,’ except that it refers to individual functions and arguments rather than to the language as a whole. At a conference on programming languages you might hear someone say, ‘The normal-order language Hassle has certain strict primitives. Other functions take their arguments by lazy evaluation.’

Ben Bitdiddle and Alyssa P. Hacker disagree over the importance of lazy evaluation for implementing things such as `unless`. Ben points out that it's possible to implement `unless` in applicative order as a new kind of expression, akin to conditional expressions. Alyssa counters that, if one did that, `unless` would be merely syntax, not a function that could be used in conjunction with higher-order functions. Fill in the details on both sides of the argument. Show how to define the evaluation of `unless` as a new kind of expression (as we defined the evaluation of conditional expressions in section 1.1.6), and give an example of a situation where it might be useful to have `unless` available as a function, rather than as a new expression syntax.

4.2.2 An Interpreter with Lazy Evaluation

In this section we will implement a normal-order language that is the same as JavaScript except that compound functions are non-strict in each argument. Primitive functions will still be strict. It is not difficult to modify the evaluator of section 4.1.1 so that the language it interprets behaves this way. Almost all the required changes center around function application.

The basic idea is that, when applying a function, the interpreter must determine which arguments are to be evaluated and which are to be delayed. The delayed arguments are not evaluated; instead, they are transformed into objects called *thunks*.²⁶

The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application. Thus, the thunk must contain the argument expression and the environment in which the function application is being evaluated.

The process of evaluating the expression in a thunk is called *forcing*.²⁷

In general, a thunk will be forced only when its value is needed: when it is passed to a primitive function that will use the value of the thunk; when it is the value of a predicate of a conditional; and when it is the value of an operator that is about to be applied as a function. One design choice we have available is whether or not to *memoize* thunks, as we did with delayed objects in section 3.5.1. With memoization, the first time a thunk is forced, it stores the value that is computed. Subsequent forcings simply return the stored value without repeating the computation. We'll make our interpreter memoize, because this is more efficient for many

²⁶The word *thunk* was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of ('thinking about') the expression could be done at compile time; thus, at run time, the expression would already have been 'thunk' about (Ingerman et al. 1960).

²⁷This is analogous to the use of force on the delayed objects that were introduced in chapter 3 to represent streams. The critical difference between what we are doing here and what we did in chapter 3 is that we are building delaying and forcing into the evaluator, and thus making this uniform and automatic throughout the language.

applications. There are tricky considerations here, however.²⁸

Modifying the evaluator

The main difference between the lazy evaluator and the one in section 4.1 is in the handling of function applications in evaluate and apply.

The is_application clause of evaluate becomes

```
is_application(exp)
? apply(actual_value(operator(exp), env),
      operands(exp),
      env)
```

This is almost the same as the is_application clause of evaluate in section 4.1.1. For lazy evaluation, however, we call apply with the operand expressions, rather than the arguments produced by evaluating them. Since we will need the environment to construct thunks if the arguments are to be delayed, we must pass this as well. We still evaluate the operator, because apply needs the actual function to be applied in order to dispatch on its type (primitive versus compound) and apply it.

Whenever we need the actual value of an expression, we use

```
function actual_value(exp, env) {
    return force_it(evaluate(exp, env));
}
```

instead of just evaluate, so that if the expression's value is a thunk, it will be forced.

Our new version of apply is also almost the same as the version in section 4.1.1. The difference is that evaluate has passed in unevaluated operand expressions: For primitive functions (which are strict), we evaluate all the arguments before applying the primitive; for compound functions (which are non-strict) we delay all the arguments before applying the function.

```
function apply(fun, args) {
    if (is_primitive_function(fun)) {
        return apply_primitive_function(
            fun, // following line changed
            list_of_arg_values(args, env));
    } else if (is_compound_function(fun)) {
        const result =
            evaluate(function_body(fun),
```

²⁸Lazy evaluation combined with memoization is sometimes referred to as *call-by-need* argument passing, in contrast to *call-by-name* argument passing. (Call-by-name, introduced in Algol 60, is similar to non-memoized lazy evaluation.) As language designers, we can build our evaluator to memoize, not to memoize, or leave this an option for programmers (exercise 4.23). As you might expect from chapter 3, these choices raise issues that become both subtle and confusing in the presence of assignments. (See exercises 4.19 and 4.21.) An excellent article by Clinger (1982) attempts to clarify the multiple dimensions of confusion that arise here.

```

        extend_environment(
            function_parameters(fun),
            // following line changed
            list_of_delayed_args(args,
                env),
            function_environment(fun)));
    if (is_return_value(result)) {
        return return_value_content(result);
    } else {
        return undefined;
    }
} else {
    Error("Unknown function type in apply",
        fun);
}
}

```

The functions that process the arguments are just like `list_of_values` from section 4.1.1, except that `list_of_delayed_args` delays the arguments instead of evaluating them, and `list_of_arg_values` uses `actual_value` instead of `evaluate`:

```

function list_of_arg_values(exps, env) {
    return no_operands(exps)
        ? null
        : pair(actual_value(first_operand(exps),
            env),
            list_of_arg_values(rest_operands(exps),
                env));
}

function list_of_delayed_args(exps, env) {
    return no_operands(exps)
        ? null
        : pair(delay_it(first_operand(exps), env),
            list_of_delayed_args(
                rest_operands(exps), env));
}

```

The other place we must change the evaluator is in the handling of `if`, where we must use `actual-value` instead of `eval` to get the value of the predicate expression before testing whether it is true or false:

```

function eval_conditional_expression(exp, env) {
    return is_true(actual_value(cond_expr_pred(exp),
        env))
        ? evaluate(cond_expr_cons(exp), env)
        : evaluate(cond_expr_alt(exp), env);
}

```

Finally, we must change the eval_toplevel function (section 4.1.4) to use actual_value instead of evaluate, so that if a delayed value is propagated back to the evaluator it will be forced before being printed.

```
function eval_toplevel(stmt) {
    const value = actual_value(stmt, the_global_environment);
    if (is_return_value(value)) {
        error("return not allowed " +
              "outside of function definitions");
    } else {
        return value;
    }
}
```

With these changes made, we can start the evaluator and test it. The successful evaluation of the **try** expression discussed in section 4.2.1 indicates that the interpreter is performing lazy evaluation:

```
eval_toplevel(parse(
    "function try(a, b) {      "
    "    return a === 0 ? 1 : b; "
    "}"
    "try(0, head(null));      "
    ));
```

Representing thunks

Our evaluator must arrange to create thunks when functions are applied to arguments and to force these thunks later. A thunk must package an expression together with the environment, so that the argument can be produced later. To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment. We use actual_value rather than evaluate so that in case the value of the expression is itself a thunk, we will force that, and so on, until we reach something that is not a thunk:

```
function force_it(obj) {
    return is_thunk(obj)
        ? actual_value(thunk_exp(obj), thunk_env(obj))
        : obj;
}
```

One easy way to package an expression with an environment is to make a list containing the expression and the environment. Thus, we create a thunk as follows:

```
function delay_it(exp, env) {
    return list("thunk", exp, env);
}
function is_thunk(obj) {
    return is_tagged_list(obj, "thunk");
```

```

}

function thunk_exp(thunk) {
    return head(tail(thunk));
}
function thunk_env(thunk) {
    return head(tail(tail(thunk)));
}

```

Actually, what we want for our interpreter is not quite this, but rather thunks that have been memoized. When a thunk is forced, we will turn it into an evaluated thunk by replacing the stored expression with its value and changing the thunk tag so that it can be recognized as already evaluated.²⁹

```

function is_evaluated_thunk(obj) {
    return is_tagged_list(obj, "evaluated_thunk");
}
function thunk_value(evaluated_thunk) {
    return head(tail(evaluated_thunk));
}
function force_it(obj) {
    if (is_thunk(obj)) {
        const result = actual_value(
            thunk_exp(obj),
            thunk_env(obj));
        set_head(obj, "evaluated_thunk");
        // replace exp with its value
        set_head(tail(obj), result);
        // forget unneeded env
        set_tail(tail(obj), null);
        return result;
    } else if(is_evaluated_thunk(obj)) {
        return thunk_value(obj);
    } else {
        return obj;
    }
}

```

Notice that the same `delay_it` function works both with and without memoization.

Exercise 4.19

Suppose we type in the following definitions to the lazy evaluator:

²⁹Notice that we also erase the env from the thunk once the expression's value has been computed. This makes no difference in the values returned by the interpreter. It does help save space, however, because removing the reference from the thunk to the env once it is no longer needed allows this structure to be *garbage-collected* and its space recycled. Similarly, we could have allowed unneeded environments in the memoized delayed objects of section 3.5.1 to be garbage-collected, by having `memo_fun` do something like `fun = null`; to discard the function `fun` (which includes the environment in which the `delay` was evaluated) after storing its value.

```
let count = 0;
function id(x) {
    count = count + 1;
    return x;
}
```

Give the missing values in the following sequence of interactions, and explain your answers.³⁰

```
read_eval_print_loop("");
// enter:      <count and id as defined above>
// response: ?
// enter:      const w = id(id(10));
// response: ?
// enter:      count
// response: ?
// enter:      w
// response: ?
// enter:      count
// response: ?
```

Exercise 4.20

The function evaluate uses actual_value rather than evaluate to evaluate the operator before passing it to apply, in order to force the value of the operator. Give an example that demonstrates the need for this forcing.

Exercise 4.21

Exhibit a program that you would expect to run much more slowly without memoization than with memoization. Also, consider the following interaction, where the id function is defined as in exercise 4.19 and count starts at 0:

```
read_eval_print_loop("");
// enter:      <count and id as defined above>
// response: ?
// enter:      function square(x) { return x * x; }
// response: ?
// enter:      square(id(10));
// response: ?
// enter:      count
// response: ?
```

Give the responses both when the evaluator memoizes and when it does not.

³⁰This exercise demonstrates that the interaction between lazy evaluation and side effects can be very confusing. This is just what you might expect from the discussion in chapter 3.

Exercise 4.22

Cy D. Fect, a reformed C programmer, is worried that some side effects may never take place, because the lazy evaluator doesn't force the expressions in a sequence. Since the value of an expression in a sequence other than the last one is not used (the expression is there only for its effect, such as assigning to a variable or printing), there can be no subsequent use of this value (e.g., as an argument to a primitive function) that will cause it to be forced. Cy thus thinks that when evaluating sequences, we must force all expressions in the sequence except the final one. He proposes to modify eval-sequence from section 4.1.1 to use actual_value rather than evaluate:

```
function eval_sequence(exps, env) {
    if (is_last_exp(exps)) {
        evaluate(first_exp(exps), env);
    } else {
        actual_value(first_exp(exps), env);
        eval_sequence(rest_exps(exps), env);
    }
}
```

- Ben Bitdiddle thinks Cy is wrong. He shows Cy the for_each function described in exercise 2.23, which gives an important example of a sequence with side effects:

```
function for_each(fun, items) {
    if (is_null(items)){
        return undefined;
    } else {
        fun(head(items));
        for_each(fun, tail(items));
    }
}
```

He claims that the evaluator in the text (with the original eval_sequence) handles this correctly:

```
read_eval_print_loop("");
// enter:      <for_each as defined above>
// response: ?
// enter:      for_each(x => display(x), list(57, 321, 88));
// response: 57
//           321
//           88
// response: done
```

Explain why Ben is right about the behavior of for_each.

- b. Cy agrees that Ben is right about the `for_each` example, but says that that's not the kind of program he was thinking about when he proposed his change to `eval_sequence`. He defines the following two functions in the lazy evaluator:

```
function p1(x) {
  x = pair(x, list(2));
}
function p2(x) {
  function p(e) {
    e;
    return x;
  }
  x = pair(x, list(2));
  return p(x);
}
```

What are the values of `p(1)` and `p2(1)` with the original `eval_sequence`? What would the values be with Cy's proposed change to `eval_sequence`?

- c. Cy also points out that changing `eval_sequence` as he proposes does not affect the behavior of the example in part a. Explain why this is true.
- d. How do you think sequences ought to be treated in the lazy evaluator? Do you like Cy's approach, the approach in the text, or some other approach?

Exercise 4.23

The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to JavaScript. It might be nicer to implement lazy evaluation as an *upward-compatible extension*, that is, so that ordinary JavaScript programs will work as before. We can do this by extending the syntax of function declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

```
function f(a, b, c, d) {
  parameters("strict", "lazy", "strict", "lazy-memo");
  ...
}
```

would define `f` to be a function of four arguments, where the first and third arguments are evaluated when the function is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary function definitions will produce the same behavior as ordinary JavaScript, while adding the "`lazy-memo`" declaration to each parameter of every compound function will produce the behavior of the lazy evaluator defined in this

section. Design and implement the changes required to produce such an extension to JavaScript. You can assume that the special ‘function call’ parameters is always the first statement in the body of a function declaration. You must also arrange for evaluate or apply to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

4.2.3 Streams as Lazy Lists

In section 3.5.1, we showed how to implement streams as delayed lists. We used a function definition expression to construct a ‘promise’ to compute the tail of a stream, without actually fulfilling that promise until later. We were forced to create streams as a new kind of data object similar but not identical to lists, and this required us to reimplement many ordinary list operations (map, append, and so on) for use with streams.

With lazy evaluation, streams and lists can be identical, so there is no need for separate list and stream operations. All we need to do is to arrange matters so that pair is non-strict. One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives, and to implement pair as one of these. An easier way is to recall (section 2.1.3) that there is no fundamental need to implement pair as a primitive at all. Instead, we can represent pairs as functions:³¹

```
function pair(x, y) {
    return m => m(x, y);
}
function head(z) {
    return z( (p, q) => p );
}
function tail(z) {
    return z( (p, q) => q );
}
```

In terms of these basic operations, the standard definitions of the list operations will work with infinite lists (streams) as well as finite ones, and the stream operations can be implemented as list operations. Here are some examples:

```
function list_ref(items, n) {
    return n === 0
        ? head(items)
        : list_ref(tail(items), n - 1);
}
function map(fun, items) {
```

³¹This is the functional representation described in exercise 2.4. Essentially any functional representation (e.g., a message-passing implementation) would do as well. Notice that we can install these definitions in the lazy evaluator simply by typing them at the driver loop. If we had originally included pair, head, and tail as primitives in the global environment, they will be redefined. (Also see exercises 4.25 and 4.26.)

```

return is_null(items)
  ? null
  : pair(fun(head(items)),
         map(fun, tail(items)));
}

function scale_list(items, factor) {
  return map(x => x * factor, items);
}

function add_lists(list1, list2) {
  return is_null(list1)
    ? list2
    : is_null(list2)
      ? list1
      : pair(head(list1) + head(list2),
             add_lists(tail(list1),
                       tail(list2)));
}

const ones = pair(1, ones);
const integers = pair(1, add_lists(ones, integers));

list_ref(integers, 17); // returns 18

```

Note that these lazy lists are even lazier than the streams of chapter 3: The head of the list, as well as the tail, is delayed.³² In fact, even accessing the head or tail of a lazy pair need not force the value of a list element. The value will be forced only when it is really needed—e.g., for use as the argument of a primitive, or to be printed as an answer.

Lazy pairs also help with the problem that arose with streams in section 3.5.4, where we found that formulating stream models of systems with loops may require us to sprinkle our programs with additional delayed function definitions, beyond the ones required to construct a stream pair. With lazy evaluation, all arguments to functions are delayed uniformly. For instance, we can implement functions to integrate lists and solve differential equations as we originally intended in section 3.5.4:

```

function integral(integrand, initial_value, dt) {
  const int =
    pair(initial_value,
          add_lists(scale_list(integrand, dt),
                    int));
  return int;
}

function solve(f, y0, dt) {
  const y = integral(dy, y0, dt);
  const dy = map(f, y);
  return y;

```

³²This permits us to create delayed versions of more general kinds of list structures, not just sequences. Hughes (1990) discusses some applications of ‘lazy trees’.

```
}
```

```
list_ref(solve(x => x, 1, 0.001), 1000);
```

Exercise 4.24

Give some examples that illustrate the difference between the streams of chapter 3 and the ‘lazier’ lazy lists described in this section. How can you take advantage of this extra laziness?

Exercise 4.25

Ben Bitdiddle tests the lazy list implementation given above by evaluating the expression

```
head(list(a, b, c));
```

To his surprise, this produces an error. After some thought, he realizes that the ‘lists’ obtained by reading in quoted expressions are different from the lists manipulated by the new definitions of pair, head, and tail. Modify the evaluator’s treatment of applications of the primitive function list typed at the driver loop will produce true lazy lists.

Exercise 4.26

Modify the driver loop for the evaluator so that lazy pairs and lists will print in some reasonable way. (What are you going to do about infinite lists?) You may also need to modify the representation of lazy pairs so that the evaluator can identify them in order to print them.

List of exercises

Exercise 1.1	31
Exercise 1.2	32
Exercise 1.3	33
Exercise 1.4	33
Exercise 1.5	33
Exercise 1.6	36
Exercise 1.7	37
Exercise 1.8	37
Exercise 1.9	47
Exercise 1.10	48

Exercise 1.11	53
Exercise 1.12	53
Exercise 1.13	53
Exercise 1.14	55
Exercise 1.15	55
Exercise 1.16	57
Exercise 1.17	58
Exercise 1.18	58
Exercise 1.19	58
Exercise 1.20	61
Exercise 1.21	64
Exercise 1.22	65
Exercise 1.23	65
Exercise 1.24	66
Exercise 1.25	66
Exercise 1.26	66
Exercise 1.27	67
Exercise 1.28	67
Exercise 1.29	71
Exercise 1.30	72
Exercise 1.31	72
Exercise 1.32	73
Exercise 1.33	73
Exercise 1.34	78
Exercise 1.35	82
Exercise 1.36	82
Exercise 1.37	83
Exercise 1.38	84
Exercise 1.39	84

Exercise 1.40	88
Exercise 1.41	89
Exercise 1.42	89
Exercise 1.43	89
Exercise 1.44	89
Exercise 1.45	90
Exercise 1.46	90
Exercise 2.1	99
Exercise 2.2	101
Exercise 2.3	101
Exercise 2.4	104
Exercise 2.5	104
Exercise 2.6	104
Exercise 2.7	106
Exercise 2.8	106
Exercise 2.9	106
Exercise 2.10	107
Exercise 2.11	107
Exercise 2.12	107
Exercise 2.13	108
Exercise 2.14	108
Exercise 2.15	109
Exercise 2.16	109
Exercise 2.17	113
Exercise 2.18	114
Exercise 2.19	114
Exercise 2.20	115
Exercise 2.21	117
Exercise 2.22	117

Exercise 2.23	118
Exercise 2.24	120
Exercise 2.25	121
Exercise 2.26	121
Exercise 2.27	122
Exercise 2.28	122
Exercise 2.29	123
Exercise 2.30	124
Exercise 2.31	125
Exercise 2.32	125
Exercise 2.33	131
Exercise 2.34	131
Exercise 2.35	132
Exercise 2.36	132
Exercise 2.37	133
Exercise 2.38	134
Exercise 2.39	135
Exercise 2.40	137
Exercise 2.41	137
Exercise 2.42	137
Exercise 2.43	139
Exercise 2.44	145
Exercise 2.45	146
Exercise 2.46	148
Exercise 2.47	148
Exercise 2.48	150
Exercise 2.49	150
Exercise 2.50	152
Exercise 2.51	152

Exercise 2.52	154
Exercise 2.53	156
Exercise 2.54	156
Exercise 2.55	156
Exercise 2.56	162
Exercise 2.57	162
Exercise 2.58	163
Exercise 2.59	165
Exercise 2.60	165
Exercise 2.61	167
Exercise 2.62	167
Exercise 2.63	170
Exercise 2.64	171
Exercise 2.65	172
Exercise 2.66	173
Exercise 2.67	179
Exercise 2.68	180
Exercise 2.69	180
Exercise 2.70	181
Exercise 2.71	181
Exercise 2.72	181
Exercise 2.73	196
Exercise 2.74	198
Exercise 2.75	200
Exercise 2.76	200
Exercise 2.77	206
Exercise 2.78	207
Exercise 2.79	207
Exercise 2.80	207

Exercise 2.81	213
Exercise 2.82	214
Exercise 2.83	214
Exercise 2.84	215
Exercise 2.85	215
Exercise 2.86	215
Exercise 2.87	222
Exercise 2.88	222
Exercise 2.89	223
Exercise 2.90	223
Exercise 2.91	223
Exercise 2.92	225
Exercise 2.93	225
Exercise 2.94	226
Exercise 2.95	227
Exercise 2.96	227
Exercise 2.97	228
Exercise 3.1	236
Exercise 3.2	237
Exercise 3.3	237
Exercise 3.4	238
Exercise 3.5	241
Exercise 3.6	242
Exercise 3.7	247
Exercise 3.8	248
Exercise 3.9	255
Exercise 3.10	260
Exercise 3.11	263
Exercise 3.12	267

Exercise 3.13	268
Exercise 3.14	268
Exercise 3.15	271
Exercise 3.16	271
Exercise 3.17	272
Exercise 3.18	272
Exercise 3.19	272
Exercise 3.20	274
Exercise 3.21	278
Exercise 3.22	278
Exercise 3.23	279
Exercise 3.24	284
Exercise 3.25	285
Exercise 3.26	285
Exercise 3.27	285
Exercise 3.28	290
Exercise 3.29	290
Exercise 3.30	291
Exercise 3.31	295
Exercise 3.32	298
Exercise 3.33	308
Exercise 3.34	308
Exercise 3.35	308
Exercise 3.36	309
Exercise 3.37	309
Exercise 3.38	316
Exercise 3.39	319
Exercise 3.40	319
Exercise 3.41	320

Exercise 3.42	320
Exercise 3.43	323
Exercise 3.44	324
Exercise 3.45	324
Exercise 3.46	327
Exercise 3.47	327
Exercise 3.48	328
Exercise 3.49	328
Exercise 3.50	337
Exercise 3.51	337
Exercise 3.52	338
Exercise 3.53	343
Exercise 3.54	343
Exercise 3.55	343
Exercise 3.56	343
Exercise 3.57	344
Exercise 3.58	345
Exercise 3.59	345
Exercise 3.60	346
Exercise 3.61	346
Exercise 3.62	347
Exercise 3.63	350
Exercise 3.64	350
Exercise 3.65	351
Exercise 3.66	353
Exercise 3.67	353
Exercise 3.68	354
Exercise 3.69	354
Exercise 3.70	354

Exercise 3.71	355
Exercise 3.72	355
Exercise 3.73	356
Exercise 3.74	357
Exercise 3.75	358
Exercise 3.76	358
Exercise 3.77	360
Exercise 3.78	361
Exercise 3.79	361
Exercise 3.80	361
Exercise 3.81	366
Exercise 3.82	366
Exercise 4.1	380
Exercise 4.2	385
Exercise 4.3	385
Exercise 4.4	391
Exercise 4.5	391
Exercise 4.6	395
Exercise 4.7	395
Exercise 4.8	398
Exercise 4.9	400
Exercise 4.10	400
Exercise 4.11	400
Exercise 4.12	401
Exercise 4.13	402
Exercise 4.14	407
Exercise 4.15	407
Exercise 4.16	408
Exercise 4.17	410

Exercise 4.18	410
Exercise 4.19	415
Exercise 4.20	416
Exercise 4.21	416
Exercise 4.22	417
Exercise 4.23	418
Exercise 4.24	421
Exercise 4.25	421
Exercise 4.26	421

Solution To Exercises

Answer of exercise 1.2

```
(5 + 4 + (2 - (3 - (6 + 4 / 5))))  
/  
(3 * (6 - 2) * (2 - 7));
```

Answer of exercise 1.3

```
function f(x, y, z) {  
    return square(x) + square(y) + square(z) -  
        // subtract the square of the smallest  
        square(x > y ? (y > z ? z : y) : (x > z ? z : x));  
}
```

Answer of exercise 1.4

According to section 1.1.5, evaluation of a application expression proceeds as follows:

- a. Evaluate the function expression of the application combination, resulting in the function to be applied.
- b. Evaluate the argument expressions of the combination.
- c. Evaluate the return expression of the function with each parameter replaced by the corresponding argument.

Thus the evaluation of the application expression `a_plus_abs_b(5, -4)` (1) evaluates `a_plus_abs_b`, resulting in the function given above, and (2) the arguments are already values. So we need to evaluate (3) the return expression of the function, with the parameters replaced by the arguments, thus: `(-4 >= 0 ? plus : minus)(5, -4)`. With the same rules, we need to (1) evaluate the function expression, which in this case is the conditional expression `-4 >= 0 ? plus : minus`. Since the predicate evaluates to *false*, the function expression evaluates to `minus`. The arguments, again (2) are already values. Thus we end up evaluating (3) the body of `minus` with the

parameters a and b replaced by 5 and -4, respectively, resulting in $5 - (-4)$, which will finally evaluate to 9.

Answer of exercise 1.5

In applicative-order evaluation of $\text{test}(0, \text{p}())$, we need to evaluate the argument expressions before we can evaluate the return expression of the function test . The evaluation of the argument expression $\text{p}()$ will not terminate, however: It will keep evaluating application expressions of the form $\text{p}()$, and thus the evalution of $\text{test}(0, \text{p}())$ will not produce a legitimate value. In normal-order evaluation, on the other hand, the function application $\text{test}(0, \text{p}())$ would immediately evaluate the return expression of the function test , $x == 0 ? 0 : y$ after replacing the parameter x with 0 and y with $\text{p}()$. The result of the replacing would be $0 == 0 ? 0 : \text{p}()$. The evaluation of the predicate $0 == 0$ results in *true* and thus the conditional expression evaluates to 0, without any need to evaluate $\text{p}()$.

Answer of exercise 1.6

Any call of sqrt_iter leads immediately to an infinite loop. The reason for this is our applicative-order evaluation. The evauation of the return expression of sqrt_iter needs to evaluate its arguments first, including the recursive call of sqrt_iter , regardless whether the predicate evaluates to *true* or *false*. The same of course happens with the recursive call, and thus the function conditional never actually gets applied.

Answer of exercise 1.7

The absolute tolerance of 0.001 is too large when computing the square root of a small value. For example, $\text{sqrt}(0.0001)$ results in 0.03230844833048122 instead of the expected value 0.01, an error of over 200%.

On the other hand, for very large values, rounding errors might make the algorithm fail to ever get close enough to the square root, in which case it will not terminate terminates.

The following program alleviates the problem by replacing an absolute tolerance with a relative tolerance.

```
const error_threshold = 0.01;
function good_enough(guess, x) {
    return relative_error(guess, improve(guess, x))
        < error_threshold;
}
function relative_error(estimate, reference) {
    return abs(estimate - reference) / reference;
}
```

Answer of exercise 1.8

```
function good_enough(guess, x) {
    return abs(cube(guess) - x) < 0.001;
}
function div3(x, y) {
    return (x + y) / 3;
}
function improve(guess, x) {
    return div3(x / (guess * guess), 2 * guess);
}
function cube_root(guess, x) {
    return good_enough(guess, x)
        ? guess
        : cuberoot(improve(guess, x), x);
}
```

Answer of exercise 1.9

The process generated by the first procedure is recursive.

```
plus(4, 5)
4 === 0 ? 5 : inc(plus(dec(4), 5))
inc(plus(dec(4), 5))
...
inc(plus(3, 5))
...
inc(inc(plus(2, 5)))
...
inc(inc(inc(plus(1, 5))))
...
inc(inc(inc(inc(plus(0, 5)))))
inc(inc(inc(inc( 0 === 0 ? 5 : inc(plus(dec(0), 5))))))
inc(inc(inc(inc( 5 ))))
inc(inc(inc( 6 )))
inc(inc( 7 ))
inc( 8 )
9
```

The process generated by the second procedure is iterative.

```
plus(4, 5)
4 === 0 ? 5 : plus(dec(4), inc(5))
plus(dec(4), inc(5))
...
plus(3, 6)
...
plus(2, 7)
...
```

```

plus(1, 8)
...
plus(0, 9)
0 === 0 ? 9 : plus(dec(0), inc(9))
9

```

Answer of exercise 1.10

The function $f(n)$ computes $2n$, the function $g(n)$ computes 2^n , and the function $h(n)$ computes $2^{2^{\dots^2}}$ where the number of 2s in the chain of exponentiation is n .

Answer of exercise 1.11

```

// iterative function
function f_iterative(n) {
    return n < 3
        ? n
        : f_iterative_impl(2, 1, 0, n - 2);
}
function f_iterative_impl(a, b, c, count) {
    return count === 0
        ? a
        : f_iterative_impl(a + 2 * b + 3 * c, a, b, count - 1);
}

//recursive function
function f_recursive(n) {
    return n < 3
        ? n
        : f_recursive(n - 1) +
            2 * f_recursive(n - 2) +
            3 * f_recursive(n - 3);
}

```

Answer of exercise 1.12

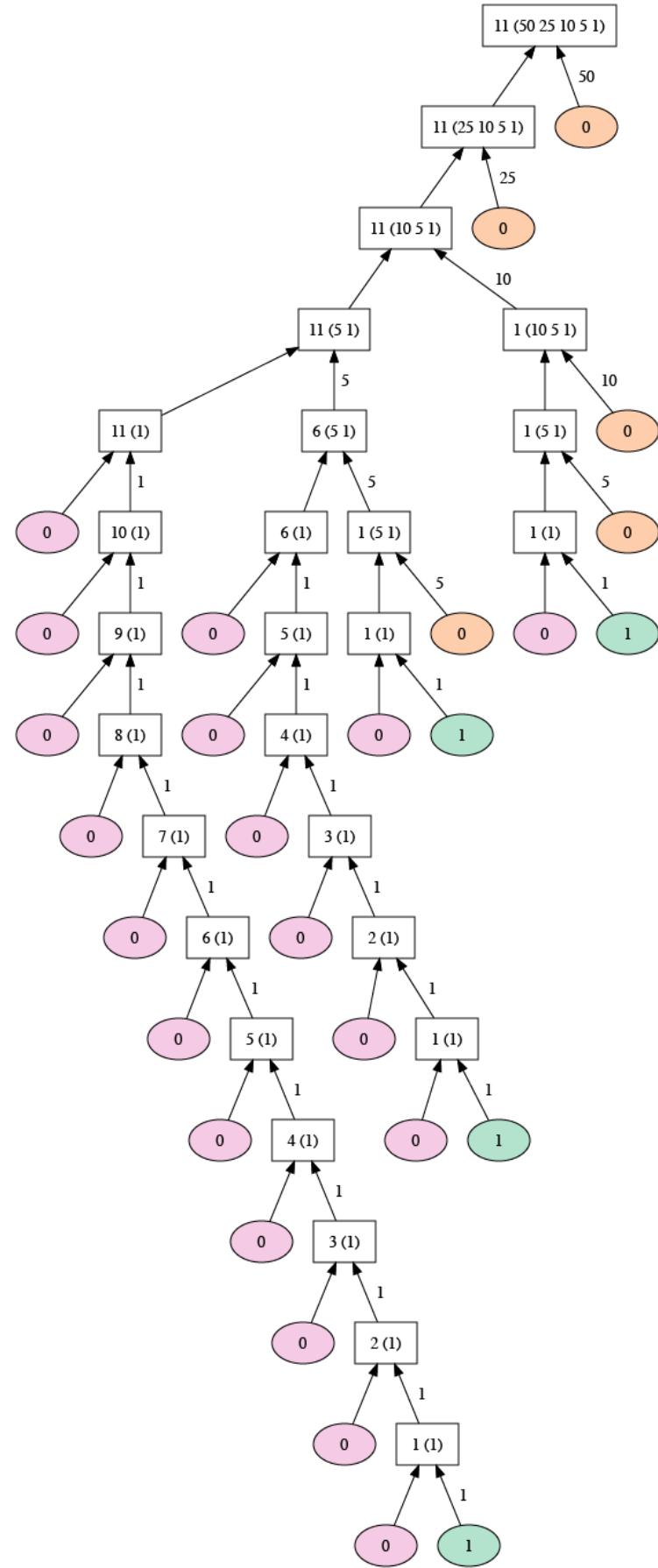
```

function pascal_triangle(row, index) {
    return index > row
        ? false
        : index === 1 || index === row
            ? 1
            : pascal_triangle(row - 1, index - 1)
                +
                pascal_triangle(row - 1, index);
}

```

Answer of exercise 1.14

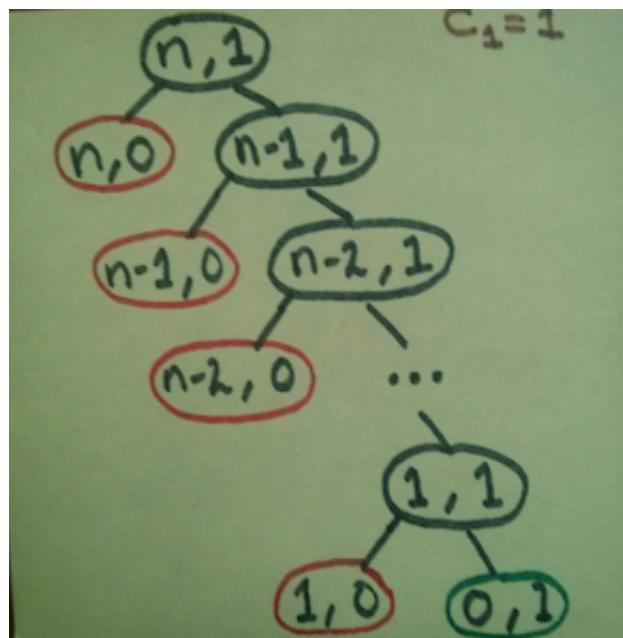
The tree-recursive process generated in computing $\text{cc}(11, 5)$ is illustrated by the image below, due to Toby Thain, assuming that the coin values in `first_denomination` are $C_1 = 1$, $C_2 = 5$, $C_3 = 10$, $C_4 = 25$ and $C_5 = 1$.



Let us consider the process for evaluating $\text{cc}(n, k)$, which means the amount to be changed is n and the number of kinds of coins is k . Let us assume the coin values are constants, not dependent on n or k .

The space required for a tree-recursive process is—as discussed in section 1.2.2—proportional to the maximum depth of the tree. At each step from a parent to a child in the tree, either n strictly decreases (by a constant coin value) or k decreases (by 1), and leaf nodes have an amount of at most 0 or a number of kinds of coins of 0. Thus, every path has a length of $\Theta(n+k)$, which is also the order of growth of the space required for $\text{cc}(n, k)$.

Let us derive a function $T(n, k)$ such that the time required for calculating $\text{cc}(n, k)$ has an order of growth of $\Theta(T(n, k))$. The following argument is due to Yati Sagade, including the illustrations (Sagade 2015). Let us start with the call tree for changing some amount n with just 1 kind of coin, i.e., the call tree for $\text{cc}(n, 1)$.

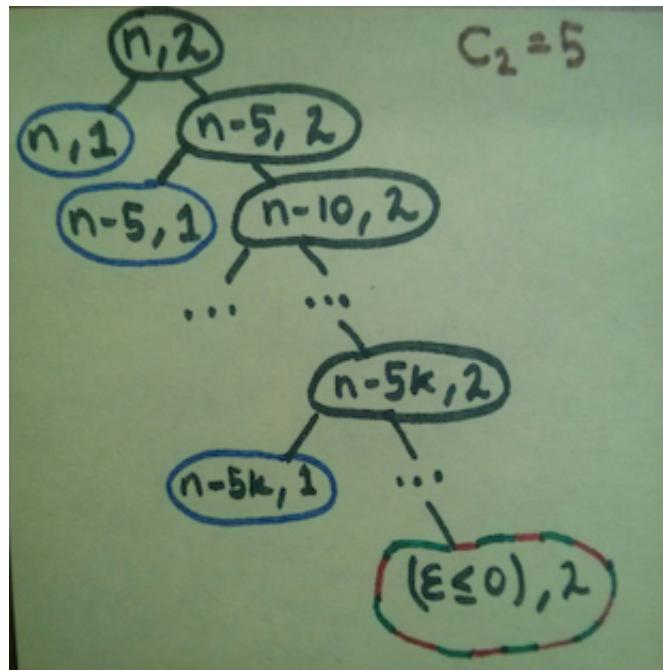


We are only allowed here to use one kind of coin, with value $C_1 = 1$. The red nodes are terminal nodes that yield 0, the green node is a terminal node that yields 1 (corresponding to the first condition in the declaration of cc). Each nonterminal node spawns two calls to cc , one (on the left) with the same amount, but fewer kinds of coins, and the other (on the right) with the amount reduced by 1 and equal kinds of coins.

Excluding the root, each level has exactly 2 nodes, and there are n such levels. This means, the number of cc calls generated by a single $\text{cc}(n, 1)$ call (including the original call) is:

$$T(n, 1) = 2n + 1 = \Theta(n)$$

Next, we will look at the call tree of $\text{cc}(n, 2)$ to calculate $T(n, 2)$:



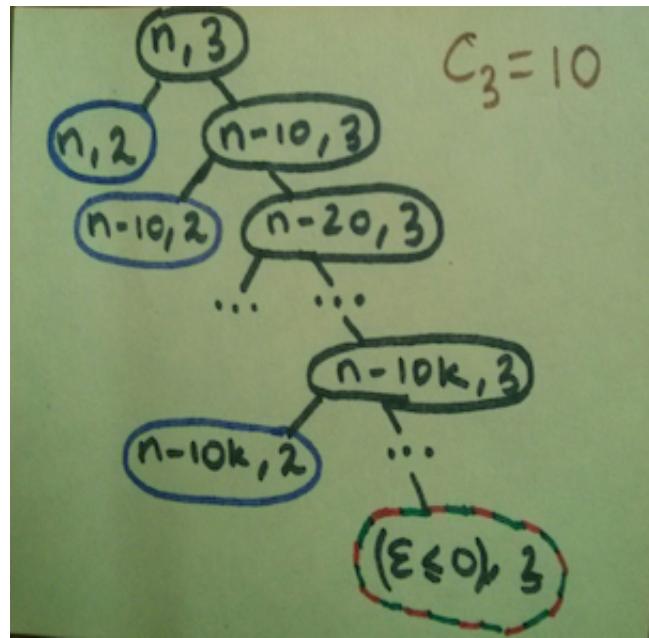
Here, we are allowed to use two denominations of coins: $\mathbb{C}_2 = 5$ and $\mathbb{C}_1 = 1$.

Each black node spawns a $cc(m, 1)$ subtree (blue), which we've already analyzed, and a $cc(m - 5, 2)$ subtree. The node colored in red and green is a terminal node, but yields 0 if the amount is less than zero and 1 if the amount is exactly zero. Sagade denotes this final amount as ϵ , which can be ≤ 0 .

Excluding the root and the last level in this tree which contains the red-green terminal node, there will be exactly $\lfloor \frac{n}{5} \rfloor$ levels. Now each of these levels contains a call to $cc(m, 1)$ (the blue nodes), each of which, in turn, is $\Theta(n)$ in time. So each of these levels contains $T(n, 1) + 1$ calls to cc . Therefore, the total number of nodes (including the terminal node and the root) in the call tree for $cc(n, 2)$ is:

$$T(n, 2) = \lfloor \frac{n}{5} \rfloor (T(n, 1) + 1) + 2 = \lfloor \frac{n}{5} \rfloor (2n + 2) + 2 = \Theta(n^2)$$

Moving ahead, let's take a look at the call tree of $cc(n, 3)$, i.e., we are now allowed to use three denominations of coins, the new addition being $\mathbb{C}_3 = 10$:



Here also, we see, similar to the previous case, that the total number of calls to cc will be

$$T(n, 3) = \lfloor \frac{n}{10} \rfloor (T(n, 2) + 1) + 2 = \lfloor \frac{n}{10} \rfloor \times \Theta(n^2) + 2 = \Theta(n^3)$$

We can see a pattern here. For some k , $k > 1$, we have,

$$T(n, k) = \lfloor \frac{n}{\mathbb{C}_k} \rfloor (T(n, k - 1) + 1) + 2$$

Here, \mathbb{C}_k is the k^{th} coin denomination. We can expand this further:

$$T(n, k) = \lfloor \frac{n}{\mathbb{C}_k} \rfloor (T(n, k - 1) + 1) + 2 = \lfloor \frac{n}{\mathbb{C}_k} \rfloor (\lfloor \frac{n}{\mathbb{C}_{k-1}} \rfloor (\dots \lfloor \frac{n}{\mathbb{C}_2} \rfloor (2n + 1) \dots)) + 2 = \Theta(n^k)$$

Note that the actual values of the coin denominations have no effect on the order of growth of this process, if we assume they are constants that do not depend on n and k .

Answer of exercise 1.15

- The function p will call itself recursively as long as the angle value is greater than 0.1. There will be altogether 5 calls of p , with arguments 12.15, 4.05, 1.35, 0.45, 0.15 and 0.05.
- The function \sin gives rise to a recursive process. In each recursive call, the angle is divided by 3 until its absolute value is smaller than 0.1. Thus the number of steps and the space required has an order of growth of $O(\log a)$. Note that the base of the logarithm is immaterial for the order of growth because the logarithms of different bases differ only by a constant factor.

Answer of exercise 1.16

```
function fast_expt_iter(a, b, n){  
    return n === 0  
        ? a  
        : is_even(n)  
            ? fast_expt_iter(a, b * b, n / 2)  
            : fast_expt_iter(a * b, b, n - 1);  
}  
function fast_expt(b, n){  
    return fast_expt_iter(1, b, n);  
}
```

Answer of exercise 1.17

```
function double(x) {  
    return x + x;  
}  
  
function halve(x) {  
    return x / 2;  
}  
function fast_times(a, b) {  
    return b === 1  
        ? a  
        : a === 0 || b === 0  
            ? 0  
            : is_even(b)  
                ? double(fast_times(a, halve(b)))  
                : a + fast_times(a, b - 1);  
}
```

Answer of exercise 1.18

```
function double(x) {  
    return x + x;  
}  
  
function half(x) {  
    return x / 2;  
}  
function fast_times_iter(total, a, b) {  
    return b === 1  
        ? total + a  
        : a === 0 || b === 0  
            ? 0  
            : is_even(b)
```

```

        ? fast_times_iter(total, double(a), half(b))
        : fast_times_iter(total + a, a, b - 1);
    }

function times(a, b) {
    return fast_times_iter(0, a, b);
}

```

Answer of exercise 1.19

```

function fib(n) {
    return fib_iter(1, 0, 0, 1, n);
}

function fib_iter(a, b, p, q, count) {
    return count === 0
        ? b
        : is_even(count)
            ? fib_iter(a,
                        b,
                        p * p + q * q,
                        2 * p * q + q * q,
                        count / 2)
            : fib_iter(b * q + a * q + a * p,
                        b * p + a * q,
                        p,
                        q,
                        count - 1);
}

```

Answer of exercise 1.20

- a. Using normal-order evaluation, the process undergoes 18 remainder operations. 14 while evaluating the condition, rest during final reduction phase.

```

gcd(206, 40)
40 === 0 ? 206 : gcd(40, 206 % 40)
gcd(40, 206 % 40)
206 % 40 === 0 ? 40 : gcd(206 % 40,
                             40 % (206 % 40))
// remainder operation (1)
6 === 0 ? 40 : gcd(206 % 40,
                     40 % (206 % 40))
gcd(206 % 40, 40 % (206 % 40))
40 % (206 % 40) === 0
? 206 % 40

```

```

        : gcd(40 % (206 % 40),
              (206 % 40) % (40 % (206 % 40)))
// remainder operations (2) and (3)
4 === 0
    ? 206 % 40
    : gcd(40 % (206 % 40),
          (206 % 40) % (40 % (206 % 40)))
gcd(40 % (206 % 40), (206 % 40) % (40 % (206 % 40)))
(206 % 40) % (40 % (206 % 40)) === 0
    ? 40 % (206 % 40)
    : gcd((206 % 40) % (40 % (206 % 40)),
          (40 % (206 % 40)) % ((206 % 40) % (40 %
                                         (206 % 40)))
// remainder operations (4), (5), (6), (7)
2 === 0
    ? 40 % (206 % 40)
    : gcd((206 % 40) % (40 % (206 % 40)),
          (40 % (206 % 40)) % ((206 % 40) % (40 %
                                         (206 % 40))))
gcd((206 % 40) % (40 % (206 % 40)),
     (40 % (206 % 40)) % ((206 % 40) % (40 % (206 % 40)))
(40 % (206 % 40)) % ((206 % 40) % (40 % (206 % 40))) === 0
    ? (206 % 40) % (40 % (206 % 40))
    : gcd((40 % (206 % 40)) % ((206 % 40) % (40 %
                                         (206 % 40)),
          ((206 % 40) % (40 % (206 % 40))) %
          ((40 % (206 % 40)) % ((206 % 40) % (40 %
                                         (206 % 40)))))
// remainder operations
    (8), (9), (10), (11), (12), (13), (14)
0 === 0
    ? (206 % 40) % (40 % (206 % 40))
    : gcd((40 % (206 % 40)) % ((206 % 40) % (40 %
                                         (206 % 40)),
          ((206 % 40) % (40 % (206 % 40))) %
          ((40 % (206 % 40)) % ((206 % 40) % (40 %
                                         (206 % 40)))))
(206 % 40) % (40 % (206 % 40))
// remainder operations (15), (16), (17), (18)
2

```

- b. Using application order evaluation, the process performs 4 remainder operations.

```

gcd(206, 40)
40 === 0 ? 206 : gcd(40, 206 % 40)
gcd(40, 206 % 40)
// remainder operation (1)
gcd(40, 6)
6 === 0 ? 40 : gcd(6, 40 % 6)

```

```

gcd(6, 40 % 6)
// remainder operation (2)
gcd(6, 4)
4 === 0 ? 6 : gcd(4, 6 % 4)
gcd(4, 6 % 4)
// remainder operation (3)
gcd(4, 2)
2 === 0 ? 4 : gcd(2, 4 % 2)
gcd(2, 4 % 2)
// remainder operation (4)
gcd(2, 0)
0 === 0 ? 2 : gcd(0, 2 % 0)
2

```

Answer of exercise 1.21

```

smallest_divisor(199);
// smallest_divisor(1999);
// smallest_divisor(19999);

```

Answer of exercise 1.25

Alyssa's suggestion is correct at first sight: her `expmod` function computes $base^{exp}$ and then finds its remainder modulo m , as required in the Fermat test.

However, for large bases, Alyssa's method will quickly bump into limitations because JavaScript uses 64 bits to represent numbers, following the double-precision floating point standard. When the numbers become so large that they cannot be represented precisely any longer in this standard, the results become unreliable. Even worse, the method might exceed the largest number that can be represented in this standard, and the computation leads to an error.

For small bases, however, Alyssa's method may be even faster than the original `expmod` function, because it will carry out only one single remainder operation.

Answer of exercise 1.26

Eva is correct: by evaluating the expression:

```

expmod(base, exp / 2, m)
* expmod(base, exp / 2, m)
% m

```

the expression `expmod(base, exp / 2, m)` is evaluated twice at each step in the computation when the exponent is even, eliminating the benefit of the fast exponentiation algorithm—which halves the exponent when the exponent is even—therefore eliminating the feature of the algorithm that makes it faster.

Answer of exercise 1.27

```
function carmichael(n) {
    function expmod(base, exp, m) {
        return exp === 0
            ? 1
            : is_even(exp)
            ? square(expmod(base, exp / 2, m)) % m
            : (base * expmod(base, exp - 1, m)) % m;
    }
    function fermat_test(n, a) {
        return expmod(a, n, n) === a;
    }
    function iter(n, i) {
        return i === n
            ? true
            : fermat_test(n, i)
            ? iter(n, i + 1)
            : false;
    }
    return iter(n, 2);
}
```

Answer of exercise 1.28

```
function random(n) {
    return math_floor(math_random() * n);
}
function miller_rabin_test(n) {
    function expmod(base, exp, m) {
        return exp === 0
            ? 1
            : is_even(exp)
            ? square(trivial_test(expmod(base,
                                         exp / 2,
                                         m),
                                         m))
            : (base * expmod(base, exp - 1, m))
            % m;
    }
    function trivial_test(r, m) {
        return r === 1 || r === m - 1
            ? r
            : square(r) % m === 1
            ? 0
            : r;
    }
}
```

```

function try_it(a) {
    return expmod(a, n - 1, n) === 1;
}
return try_it(1 + random(n - 1));
}

function do_miller_rabin_test(n, times) {
    return times === 0
        ? true
        : miller_rabin_test(n)
            ? do_miller_rabin_test(n, times - 1)
            : false;
}

```

Answer of exercise 1.29

```

function inc(k) {
    return k + 1;
}
function simpsons_rule_integral(f, a, b, n) {
    function helper(h) {
        function y(k) {
            return f((k * h) + a);
        }
        function term(k) {
            return k === 0 || k === n
                ? y(k)
                : k % 2 === 0
                    ? 2 * y(k)
                    : 4 * y(k);
        }
        return sum(term, 0, inc, n) * (h / 3);
    }
    return helper((b - a) / n);
}

```

Answer of exercise 1.30

```

function sum(term, a, next, b) {
    function iter(a, result) {
        return a > b
            ? result
            : iter(next(a), result + term(a));
    }
    return iter(a, 0);
}

```

Answer of exercise 1.31

```
//recursive process
function product_r(term, a, next, b) {
    return a > b
        ? 1
        : term(a) * product_r(term, next(a), next, b);
}

//iterative process
function product_i(term, a, next, b) {
    function iter(a, result) {
        return a > b
            ? result
            : iter(next(a), term(a) * result);
    }
    return iter(a, 1);
}
```

Answer of exercise 1.32

```
//recursive process
function accumulate_r(combiner, null_value, term, a, next, b) {
    return a > b
        ? null_value
        : combiner(term(a),
                    accumulate_r(combiner,
                                null_value,
                                term, next(a), next, b));
}

//iterative process
function accumulate_i(combiner, null_value, term, a, next, b) {
    function iter(a, result) {
        return a > b
            ? result
            : iter(next(a), combiner(term(a), result));
    }
    return iter(a, null_value);
}

function sum_i(term, a, next, b) {
    function plus(x, y) {
        return x + y;
    }
    return accumulate_i(plus, 0, term, a, next, b);
}

function product_r(term, a, next, b) {
    function times(x, y) {
        return x * y;
```

```

    }
    return accumulate_r(times, 1, term, a, next, b);
}

```

Answer of exercise 1.33

```

function filtered_accumulate(combiner, null_value,
                             term, a, next, b, filter) {
  return a > b
    ? null_value
    : filter(a)
      ? combiner(term(a),
                  filtered_accumulate(combiner, null_value,
                                      term, next(a), next,
                                      b, filter))
      : filtered_accumulate(combiner, null_value,
                            term, next(a), next,
                            b, filter);
}

```

Answer of exercise 1.34

Let's use the substitution model to illustrate what happens: The application combination 2(2) leads to an error, since 2 is neither a primitive nor a compound function.

Answer of exercise 1.35

The fixed point of the function is

$$1 + 1/x = x$$

Solving for x , we get

$$x^2 = x + 1$$

$$x^2 - x - 1 = 0$$

Using the quadratic equation to solve for x , we find that one of the roots of this equation is the golden ratio $(1 + \sqrt{5})/2$.

```
fixed_point(x => 1 + (1 / x), 1.0);
```

Answer of exercise 1.36

We modify the function `fixed_point` as follows:

```

const tolerance = 0.00001;
function fixed_point(f, first_guess) {
    function close_enough(x, y) {
        return abs(x - y) < tolerance;
    }
    function try_with(guess) {
        display(guess);
        const next = f(guess);
        return close_enough(guess, next)
            ? next
            : try_with(next);
    }
    return try_with(first_guess);
}

```

Here is a version with average dampening built-in:

```

function fixed_point_with_average_dampening(f, first_guess) {
    function close_enough(x, y) {
        return abs(x - y) < tolerance;
    }
    function try_with(guess) {
        display(guess);
        const next = (guess + f(guess)) / 2;
        return close_enough(guess, next)
            ? next
            : try_with(next);
    }
    return try_with(first_guess);
}

```

Answer of exercise 1.37

```

//recursive process
function cont_frac(n, d, k) {
    function fraction(i) {
        return i > k
            ? 0
            : n(i) / (d(i) + fraction(i + 1));
    }
    return fraction(1);
}

```

```

//iterative process
function cont_frac(n, d, k) {
    function fraction(i, current) {
        return i === 0
            ? current
            : fraction(i - 1, n(i) / (d(i) + current));
    }
    return fraction(k, 0);
}

cont_frac(i => 1.0,
          i => 1.0,
          20);

```

Answer of exercise 1.38

```

2 + cont_frac(i => 1,
               i => (i + 1) % 3 < 1 ? 2 * (i + 1) / 3 : 1,
               20);

```

Answer of exercise 1.39

```

function tan_cf(x, k) {
    return cont_frac(i => i === 1 ? x : -x * x,
                     i => 2 * i - 1,
                     k);
}

```

Answer of exercise 1.40

```

function cubic(a, b, c) {
    return x => cube(x) + a * square(x) + b * x + c;
}

```

Answer of exercise 1.41

```

function double(f) {
    return x => f(f(x));
}

```

Answer of exercise 1.42

```

function compose(f, g) {
    return x => f(g(x));
}

```

Answer of exercise 1.43

```
function repeated(f, n) {
    return n === 0
        ? x => x
        : compose(f, repeated(f, n - 1));
}
```

Answer of exercise 1.44

```
const dx = 0.00001;
function smooth(f) {
    return x => (f(x - dx) + f(x) + f(x + dx)) / 3;
}
function n_fold_smooth(f, n) {
    return repeated(smooth, n)(f);
}
```

Answer of exercise 1.45

```
function nth_root(n, x) {
    return fixed_point(repeated(average_damp,
                                math_floor(math_log2(n)))
                      (y => x / fast_expt(y, n - 1)),
                      1.0);
}
```

Answer of exercise 1.46

```
function iterative_improve(good_enough, improve) {
    function iterate(guess) {
        return good_enough(guess)
            ? guess
            : iterate(improve(guess));
    }
    return iterate;
}

function sqrt(x) {
    return iterative_improve(y => good_enough(y, x),
                            y => improve(y, x))(1);
}

sqrt(49);
```

```

const tolerance = 0.00001;
function fixed_point(f, first_guess) {
    return iterative_improve(
        guess => abs(guess - f(guess)) < tolerance,
        f)
    (first_guess);
}

fixed_point(math_cos, 1.0);

```

Answer of exercise 2.1

```

function sign(x) {
    return x < 0
    ? -1
    : x > 0
    ? 1
    : 0;
}
function make_rat(n, d) {
    const g = gcd(n, d);
    return pair(sign(n) * sign(d) * abs(n / g),
              abs(d / g));
}

```

Answer of exercise 2.2

```

function x_point(x) {
    return head(x);
}
function y_point(x) {
    return tail(x);
}
function make_point(x, y) {
    return pair(x, y);
}
function make_segment(start_point, end_point) {
    return pair(start_point, end_point);
}
function start_segment(x) {
    return head(x);
}
function end_segment(x) {
    return tail(x);
}
function average(a, b) {
    return (a + b) / 2;
}

```

```

}

function mid_point_segment(x) {
    const a = start_segment(x);
    const b = end_segment(x);
    return make_point(average(x_point(a),
                                x_point(b)),
                      average(y_point(a),
                                y_point(b)));
}

```

Answer of exercise 2.3

First implementation:

```

function make_point(x,y){
    return pair(x,y);
}

function x_point(x){
    return head(x);
}

function y_point(x){
    return tail(x);
}

function make_rect(bottom_left, top_right){
    return pair(bottom_left, top_right);
}

function top_right(rect){
    return tail(rect);
}

function bottom_right(rect){
    return make_point(x_point(tail(rect)),
                      y_point(head(rect))));
}

function top_left(rect){
    return make_point(x_point(head(rect)),
                      y_point(tail(rect)));
}

function bottom_left(rect){
    return head(rect);
}

function abs(x){
    return x < 0 ? -x : x;
}

```

```

}

function width_rect(rect){
    return abs(x_point(bottom_left(rect)) -
               x_point(bottom_right(rect)));
}

function height_rect(rect){
    return abs(y_point(bottom_left(rect)) -
               y_point(top_left(rect)));
}

function area_rect(rect){
    return width_rect(rect) * height_rect(rect);
}

function perimeter_rect(rect){
    return 2 * (width_rect(rect) + height_rect(rect));
}

```

Second implementation:

```

function make_point(x,y){
    return pair(x,y);
}

function make_rect(bottom_left, width, height){
    return pair(bottom_left, pair(width, height));
}

function height_rect(rect){
    return tail(tail(rect));
}

function width_rect(rect){
    return head(tail(rect));
}

function area_rect(rect){
    return width_rect(rect) * height_rect(rect);
}

function perimeter_rect(rect){
    return 2 * (width_rect(rect) + height_rect(rect));
}

```

Answer of exercise 2.4

```

function tail(z) {
    return z((p,q) => q);
}

```

Answer of exercise 2.5

```

function pair(a, b) {
    return fast_expt(2, a) * fast_expt(3, b);
}
function head(p) {
    return p % 2 === 0
        ? head(p / 2) + 1
        : 0;
}
function tail(p) {
    return p % 3 === 0
        ? tail(p/3) + 1
        : 0;
}

```

Answer of exercise 2.6

```

function one() {
    return f => x => f(x);
}
function two() {
    return f => x => f(f(x));
}
function plus(n, m) {
    return f => x => n(f)(m(f))(x);
}

```

Answer of exercise 2.7

```

function make_interval(x, y) {
    return pair;
}
function lower_bound(x) {
    return head(x);
}
function upper_bound(x) {
    return tail(x);
}

```

Answer of exercise 2.8

```

function sub_interval(x, y) {
    return make_interval(lower_bound(x) - upper_bound(y),
                         upper_bound(x) - lower_bound(y));
}

```

Answer of exercise 2.9

Let us denote the width of interval i with $W(i)$, and its lower and upper bound with $L(i)$ and $U(i)$, respectively. Two intervals i_1 and i_2 have by definition widths of $(U(i_1) - L(i_1))/2$ and $(U(i_2) - L(i_2))/2$, respectively. Adding the two intervals leads to the interval $[L(i_1) + L(i_2), U(i_1) + U(i_2)]$, whose width is

$$\begin{aligned}
 & (U(i_1) + U(i_2) - (L(i_1) + L(i_2)))/2 \\
 &= (U(i_1) - L(i_1))/2 + (U(i_2) - L(i_2))/2 \\
 &= (W(i_1) + W(i_2))
 \end{aligned}$$

The argument for subtraction is similar.

The widths of the result of multiplying intervals does not have such a nice property. For example, multiplying any interval with the zero-width interval $[0, 0]$ yields a zero-width interval whereas multiplying any interval i with the zero-width interval $[1, 1]$ yields an interval with width $W(i)$. The argument for division is similar.

Answer of exercise 2.10

```

function div_interval(x, y) {
    return lower_bound(y) <= 0 && upper_bound(y) >= 0
        ? error("Division error (interval spans 0)")
        : mul_interval(x, make_interval(1 / upper_bound(y),
                                         1 / lower_bound(y)));
}

```

Answer of exercise 2.11

```

function p(n) {
    return n >= 0;
}
function n(n) {
    return ! p(n);
}
function the_trouble_maker(xl, xu, yl, yu) {
    const p1 = xl * yl;
    const p2 = xl * yu;
    const p3 = xu * yl;
    const p4 = xu * yu;
    make_interval(math_min(p1, p2, p3, p4),

```

```

        math_max(p1, p2, p3, p4));
}

function mul_interval(x, y) {
    const xl = lower_bound(x);
    const xu = upper_bound(x);
    const yl = lower_bound(y);
    const yu = upper_bound(y);
    return p(xl) && p(xu) && p(yl) && p(yu)
        ? make_interval(xl * yl, xu * yu)
        : p(xl) && p(xu) && n(yl) && p(yu)
            ? make_interval(xu * yl, xu * yu)
            : p(xl) && p(xu) && n(yl) && n(yu)
                ? make_interval(xu * yl, xl * yu)
                : n(xl) && p(xu) && p(yl) && p(yu)
                    ? make_interval(xl * yu, xu * yu)
                    : n(xl) && p(xu) && n(yl) && n(yu)
                        ? make_interval(xu * yl, xl * yl)
                        : n(xl) && n(xu) && p(yl) && p(yu)
                            ? make_interval(xl * yu, xu * yl)
                            : n(xl) && n(xu) && n(yl) && p(yu)
                                ? make_interval(xl * yu, xl * yl)
                                : n(xl) && n(xu) && n(yl) && n(yu)
                                    ? make_interval(xu * yu, xl * yl)
                                    : n(xl) && p(xu) && n(yl) && p(yu)
                                        ? the_trouble_maker(xl, xu, yl, yu)
                                        : error("lower larger than upper");
}

```

Answer of exercise 2.12

```

function make_center_percent(center, percent) {
    const width = center * (percent / 100);
    return make_center_width(center, width);
}
function percent(i) {
    return (width(i) / center(i)) * 100;
}

```

Answer of exercise 2.13

Let us denote the maximal error of an interval with center i by Δi the maximal error of an interval with center j by Δj , and the maximal error of the multiplication result with center k by Δk . Then:

$$k + \Delta k = (i + \Delta i) * (j + \Delta j) = ij + j\Delta i + i\Delta j + \Delta i \Delta j$$

Since $k = ij$

$$\Delta k = j\Delta i + i\Delta j + \Delta i \Delta j$$

Since we assume that $\Delta i \ll i$ and $\Delta j \ll j$, we can neglect the term $\Delta i \Delta j$ and obtain

$$\Delta k = j\Delta i + i\Delta j$$

Expressed in tolerances, we obtain:

$$\Delta k/k = (j\Delta i + i\Delta j)/ij = \Delta i/i + \Delta j/j$$

Thus, the tolerance of the result of an interval multiplication is (roughly) the sum of the tolerances of its arguments.

Answer of exercise 2.14

The expression A/A is interesting, because if the interval is meant to represent a specific (albeit imprecisely known) value, the result should be exactly 1 (width 0), whereas the interval division will result in an interval with positive width. Multiple occurrences of the same term are not recognized as such in the approaches above and thus they will suffer from this problem.

Answer of exercise 2.15

She is right. The so-called *dependency problem* in interval arithmetic arises when the same input values (or intermediate terms) appear in a function on intervals. The second formulation is better because each input occurs only once, and therefore the result of a naive interval calculation is optimal.

Answer of exercise 2.16

The dependency problem in interval arithmetic is solved using linear and polynomial approximations, leading to affine arithmetic and Taylor series methods, respectively.

Answer of exercise 2.17

```
function last_pair(items) {
    return is_null(tail(items))
        ? items
        : last_pair(tail(items));
}
```

Answer of exercise 2.18

```

// naive reverse (what is the runtime?)
function reverse(items) {
    return is_null(items)
        ? null
        : append(reverse(tail(items)),
                  pair(head(items), null));
}

// a better version
function reverse(items) {
    function reverse_iter(items, result) {
        return is_null(items)
            ? result
            : reverse_iter(tail(items),
                           pair(head(items), result));
    }
    return reverse_iter(items, null);
}

```

Answer of exercise 2.19

```

function first_denomination(coin_values) {
    return head(coin_values);
}
function except_first_denomination(coin_values) {
    return tail(coin_values);
}
function no_more(coin_values) {
    return is_null(coin_values);
}

```

The order of the list `coin_values` does not affect the answer given by any correct solution of the problem, because the given list represents an unordered collection of denominations.

Answer of exercise 2.20

a.

```

function brooks(f, items) {
    return is_null(items)
        ? f
        : brooks(f(head(items)), tail(items));
}

```

b.

```

function brooks_curried(items) {
    return brooks(head(items), tail(items));
}

```

c. The statement

```
brooks_curried(list(brooks_curried,
                     list(plus_curried, 3, 4)));
```

of course evaluates to 7, as does

d.

```
brooks_curried(list(brooks_curried,
                     list(brooks_curried,
                           list(plus_curried, 3, 4))));
```

Answer of exercise 2.21

```
function square_list(items) {
  return is_null(items)
    ? null
    : pair(square(head(items)),
           square_list(tail(items)));
}

function square_list(items) {
  return map(square, items);
}
```

Answer of exercise 2.22

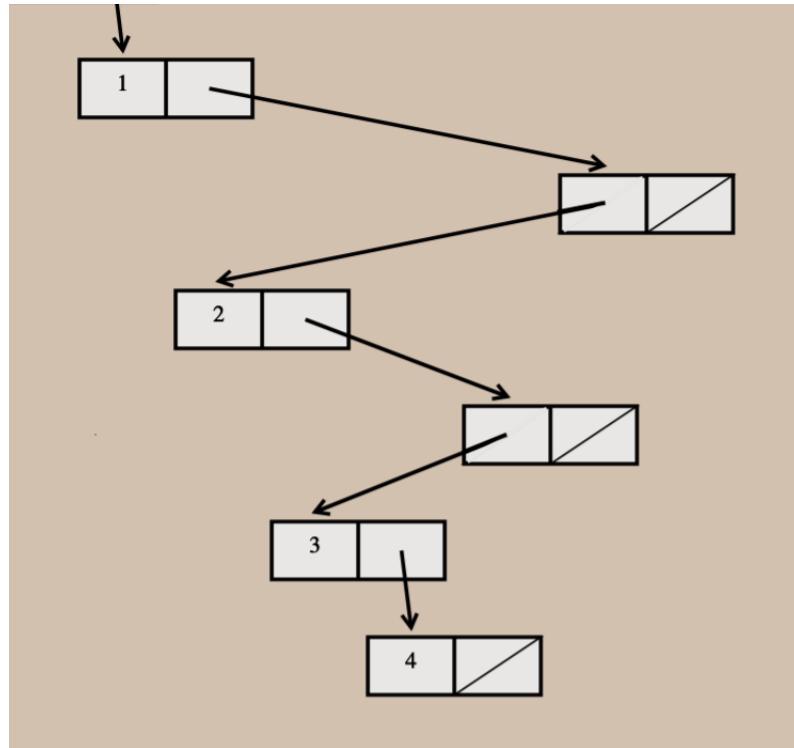
- The result list is reversed in the first program because the argument list is traversed in the given order, from first to last, but squares are added successively to the front of the answer list via `pair`. The last element of the list is the last one to be added to the answer and thus ends up as the first element of the result list.
- The second program makes things worse! The result is not even a list any longer, because the elements occupy the tail position of the result list and not the head position.

Answer of exercise 2.23

```
function for_each(fun, items) {
  if (is_null(items)){
    return undefined;
  } else {
    fun(head(items));
    for_each(fun, tail(items));
  }
}
```

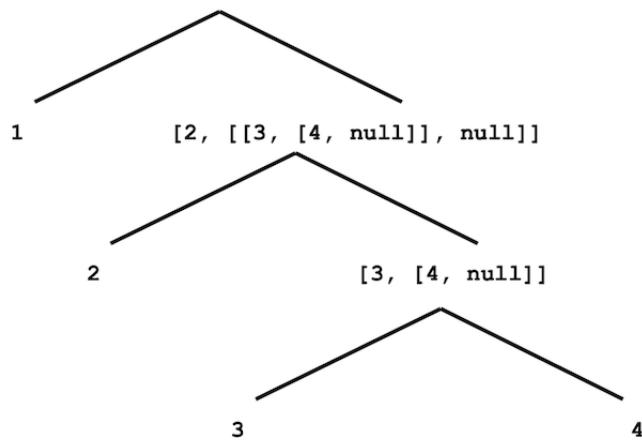
Answer of exercise 2.24

a. `[1, [[2, [[3, [4, null]], null]], null]]`



b.

`[1, [[2, [[3, [4, null]], null]], null]]`



c.

Answer of exercise 2.25

```
- head(tail(head(tail(tail(the_first_list))))));  
  
- head(head(the_second_list));  
  
- head(tail(head(tail(head(tail(head(tail(head(tail(head(tail(head(tail(head(the_third_list))))))))))))));
```

Answer of exercise 2.26

- a.

```
[1, [2, [3, [4, [5, [6, null]]]]]]]
```
- b.

```
[[1, [2, [3, null]]], [4, [5, [6, null]]]]]
```
- c.

```
[[1, [2, [3, null]]], [[4, [5, [6, null]]], null]]]
```

Answer of exercise 2.27

```
function deep_reverse(items){  
    return is_null(items)  
        ? null  
        : is_pair(items)  
            ? append(deep_reverse(tail(items)),  
                    pair(deep_reverse(head(items)),  
                          null))  
            : items;  
}
```

Answer of exercise 2.28

```
function fringe(x) {  
    return is_null(x)  
        ? null  
        : is_pair(x)  
            ? append(fringe(head(x)), fringe(tail(x)))  
            : list(x);  
}
```

Answer of exercise 2.29

a.

```
function left_branch(m) {
    return head(m);
}
function right_branch(m) {
    return head(tail(m));
}
function branch_length(b) {
    return head(b);
}
function branch_structure(b) {
    return head(tail(b));
}
```

b.

```
function is_weight(x){
    return is_number(x);
}
function total_weight(x) {
    return is_weight(x)
        ? x
        : total_weight(branch_structure(
            left_branch(x))) +
        total_weight(branch_structure(
            right_branch(x)));
}
```

c.

```
function is_balanced(x) {
    return is_weight(x) ||
        ( is_balanced(branch_structure(
            left_branch(x))) &&
        is_balanced(branch_structure(
            right_branch(x))) &&
        total_weight(branch_structure(
            left_branch(x)))
        * branch_length(left_branch(x))
        ===
        total_weight(branch_structure(
            right_branch(x)))
        * branch_length(right_branch(x)))
    );
}
```

d. With this alternative representation, the accessor functions for mobile and branch need to change as follows:

```
function left_branch(m) {
    return head(m);
```

```

    }
  function right_branch(m) {
    return tail(m);
}
function branch_length(b) {
  return head(b);
}
function branch_structure(b) {
  return tail(b);
}

```

Answer of exercise 2.30

Directly:

```

function square_tree(tree) {
  return is_null(tree)
    ? null
    : ! is_pair(tree)
      ? square(tree)
      : pair(square_tree(head(tree)),
              square_tree(tail(tree)));
}

```

The version using map:

```

function square_tree(tree) {
  return map(sub_tree => ! is_pair(sub_tree)
            ? square(sub_tree)
            : square_tree(sub_tree),
            tree);
}

```

Answer of exercise 2.31

```

function tree_map(f, tree) {
  return map(sub_tree => is_null(sub_tree)
            ? null
            : is_pair(sub_tree)
              ? tree_map(f, sub_tree)
              : f(sub_tree),
            tree);
}

```

Answer of exercise 2.32

```

function subsets(s) {
  if (is_null(s)) {
    return list(null);
  } else {
    const rest = subsets(tail(s));
    return append(rest, map(x => pair(head(s), x), rest));
  }
}

```

The argument starts in a similar way as the argument for the function `cc` in section 1.2.2: A subset either contains the first element e of the given set, or it doesn't. If it doesn't, the problem becomes strictly smaller: Compute all subsets of the tail of the list that represents the given set. If it does, it must result from adding e to a subset that doesn't contain e . In the end, we need to append both lists of subsets to obtain the list of all subsets.

Answer of exercise 2.33

```

function map(f, sequence) {
  return accumulate((x, y) => pair(f(x), y),
                    null,
                    sequence);
}

function append(seq1, seq2) {
  return accumulate(pair, seq2, seq1);
}

function length(sequence) {
  return accumulate((x, y) => y + 1,
                    0,
                    sequence);
}

```

Answer of exercise 2.34

```

function horner_eval(x, coefficient_sequence) {
  return accumulate((this_coeff, higher_terms) =>
                    x * higher_terms + this_coeff,
                    0,
                    coefficient_sequence);
}

```

Answer of exercise 2.35

```

function count_leaves(t) {
    return accumulate((leaves, total) => leaves + total,
                      0,
                      map(sub_tree => is_pair(sub_tree)
                           ? count_leaves(sub_tree)
                           : 1,
                           t));
}

```

Answer of exercise 2.36

```

function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
        ? null
        : pair(accumulate(op, init, map(x => head(x), seqs)),
               accumulate_n(op, init, map(x => tail(x), seqs)));
}

```

Answer of exercise 2.37

```

function matrix_times_vector(m, v) {
    return map(row => dot_product(row, v), m);
}

function transpose(mat) {
    return accumulate_n(pair, null, mat);
}
function matrix_times_matrix(n, m) {
    const cols = transpose(m);
    return map(x => map(y => dot_product(x, y), cols), n);
}

```

Answer of exercise 2.38

We can guarantee that `fold_right` and `fold_left` produce the same values for any sequence, if we require that *op* is commutative and associative.

```

fold_right(plus, 0, list(1, 2, 3));
fold_left(plus, 0, list(1, 2, 3));

```

Answer of exercise 2.39

```

function reverse(sequence) {
    return fold_right((x, y) => append(y, list(x)),
                      null, sequence);
}

```

```

function reverse(sequence) {
    return fold_left((x, y) => pair(y, x), null, sequence);
}

```

Answer of exercise 2.40

```

function unique_pairs(n) {
    return flatmap(i => map(j => list(i, j),
                           enumerate_interval(1, i-1)),
                  enumerate_interval(1, n));
}

function prime_sum_pairs(n) {
    return map(make_pair_sum,
               filter(is_prime_sum,
                      unique_pairs(n)));
}

```

Answer of exercise 2.41

```

function unique_triples(n) {
    return flatmap(i => flatmap(j => map(k => list(i, j, k),
                                              enumerate_interval(1, j-1)),
                                 enumerate_interval(1, i-1)),
                  enumerate_interval(1, n));
}

function plus(x, y) {
    return x + y;
}

function triples_that_sum_to(s, n) {
    return filter(items => accumulate(plus, 0, items) === s,
                  unique_triples(n));
}

```

Answer of exercise 2.42

```

function adjoin_position(row, col, rest) {
    return pair(pair(row, col), rest);
}

const empty_board = null;

function is_safe(k, positions) {
    const first_row = head(head(positions));
    const first_col = tail(head(positions));
    return accumulate((pos, so_far) => {
        const row = head(pos);
        const col = tail(pos);

```

```

        return so_far &&
               first_row - first_col !==
               row - col &&
               first_row + first_col !==
               row + col &&
               first_row !== row;
    },
    true,
    tail(positions));
}

```

Putting it all together:

```
// click here to see the solution
```

Answer of exercise 2.43

Louis's program re-evaluates the application `queen_cols(k - 1)` in each iteration of `flatmap`, which happens n times for each k . That means overall Louis's program will solve the puzzle in a time of about $n^n T$ if the program in exercise 2.42 solves the puzzle in time T .

Answer of exercise 2.44

```

function up_split(painter, n) {
  if (n === 0) {
    return painter;
  } else {
    const smaller = up_split(painter, n - 1);
    return stack(beside(smaller, smaller), painter);
  }
}

```

Answer of exercise 2.45

```

function split(identity_op, smaller_op) {
  function rec_split(painter, n) {
    if (n === 0) {
      return painter;
    } else {
      const smaller = rec_split(painter, n - 1);
      return identity_op(painter,
                         smaller_op(smaller, smaller));
    }
  }
  return rec_split;
}

```

```

const right_split = split(beside, stack);

show(right_split(heart, 4));

```

Answer of exercise 2.46

```

function make_vect(x, y) {
    return pair(x, y);
}
function xcor_vect(vector) {
    return head(vector);
}
function ycor_vect(vector) {
    return tail(vector);
}
function scale_vect(factor, vector) {
    return make_vect(factor * xcor_vect(vector),
                    factor * ycor_vect(vector));
}
function add_vect(vector1, vector2) {
    return make_vect(xcor_vect(vector1)
                     + xcor_vect(vector2),
                     ycor_vect(vector1)
                     + ycor_vect(vector2));
}
function sub_vect(vector1, vector2) {
    return make_vect(xcor_vect(vector1)
                     - xcor_vect(vector2),
                     ycor_vect(vector1)
                     - ycor_vect(vector2));
}

```

Answer of exercise 2.47

a.

```

function make_frame(origin, edge1, edge2) {
    return list(origin, edge1, edge2);
}
function origin_frame(frame) {
    return list_ref(frame, 0);
}
function edge1_frame(frame) {
    return list_ref(frame, 1);
}
function edge2_frame(frame) {
    return list_ref(frame, 2);
}

```

b.

```
function make_frame(origin, edge1, edge2) {
    return pair(origin, pair(edge1, edge2));
}
function origin_frame(frame) {
    return head(frame);
}
function edge1_frame(frame) {
    return head(tail(frame));
}
function edge2_frame(frame) {
    return tail(tail(frame));
}
```

Answer of exercise 2.48

```
function make_segment(v_start, v_end) {
    return pair(v_start, v_end);
}
function start_segment(v) {
    return head(v);
}
function end_segment(v) {
    return tail(v);
}
```

Answer of exercise 2.49

a. The painter that draws the outline of the designated frame.

```
const outline_start_1 = make_vect(0.0, 0.0);
const outline_end_1 = make_vect(1.0, 0.0);
const outline_segment_1 = make_segment(outline_start_1,
                                         outline_end_1);
const outline_start_2 = make_vect(1.0, 0.0);
const outline_end_2 = make_vect(1.0, 1.0);
const outline_segment_2 = make_segment(outline_start_2,
                                         outline_end_2);
const outline_start_3 = make_vect(1.0, 1.0);
const outline_end_3 = make_vect(0.0, 1.0);
const outline_segment_3 = make_segment(outline_start_3,
                                         outline_end_3);
const outline_start_4 = make_vect(0.0, 1.0);
const outline_end_4 = make_vect(0.0, 0.0);
const outline_segment_4 = make_segment(outline_start_4,
                                         outline_end_4);
```

```

const outline_painter = segments_to_painter(
    list(outline_segment_1,
        outline_segment_2,
        outline_segment_3,
        outline_segment_4));

```

- b. The painter that draws an ‘X’ by connecting opposite corners of the frame.

```

const x_start_1 = make_vect(0.0, 0.0);
const x_end_1 = make_vect(1.0, 1.0);
const x_segment_1 = make_segment(x_start_1,
    x_end_1);
const x_start_2 = make_vect(1.0, 0.0);
const x_end_2 = make_vect(0.0, 1.0);
const x_segment_2 = make_segment(x_start_2,
    x_end_2);
const x_painter = segments_to_painter(
    list(x_segment_1,
        x_segment_2));

```

- c. The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.

```

const diamond_start_1 = make_vect(0.5, 0.0);
const diamond_end_1 = make_vect(1.0, 0.5);
const diamond_segment_1 = make_segment(diamond_start_1,
    diamond_end_1);
const diamond_start_2 = make_vect(1.0, 0.5);
const diamond_end_2 = make_vect(0.5, 1.0);
const diamond_segment_2 = make_segment(diamond_start_2,
    diamond_end_2);
const diamond_start_3 = make_vect(0.5, 1.0);
const diamond_end_3 = make_vect(0.0, 0.5);
const diamond_segment_3 = make_segment(diamond_start_3,
    diamond_end_3);
const diamond_start_4 = make_vect(0.0, 0.5);
const diamond_end_4 = make_vect(0.5, 0.0);
const diamond_segment_4 = make_segment(diamond_start_4,
    diamond_end_4);
const diamond_painter = segments_to_painter(
    list(diamond_segment_1,
        diamond_segment_2,
        diamond_segment_3,
        diamond_segment_4));

```

Answer of exercise 2.50

a. The transformation `flip_horiz`:

```
function flip_horiz(painter) {
    return transform_painter(painter,
        make_vect(1.0, 0.0), // new origin
        make_vect(0.0, 0.0), // new end of edge1
        make_vect(1.0, 1.0)); // new end of edge2
}
```

b. The transformation `rotate180`:

```
function rotate180(painter) {
    return transform_painter(
        painter,
        make_vect(1.0, 1.0), // new origin
        make_vect(0.0, 1.0), // new end of edge1
        make_vect(1.0, 0.0)); // new end of edge2
}
```

c. The transformation `rotate270`:

```
function rotate270(painter) {
    return transform_painter(
        painter,
        make_vect(0.0, 1.0), // new origin
        make_vect(0.0, 0.0), // new end of edge1
        make_vect(1.0, 0.0)); // new end of edge2
}
```

Answer of exercise 2.51

a. First the direct method:

```
function stack(painter1, painter2) {
    const split_point = make_vect(0.0, 0.5);
    const paint_upper =
        transform_painter(painter1,
            split_point,
            make_vect(1.0, 0.5),
            make_vect(0.0, 1.0));
    const paint_lower =
        transform_painter(painter2,
            make_vect(0.0, 0.0),
            make_vect(1.0, 0.0),
            split_point);
    return frame => {
        paint_upper(frame);
        paint_lower(frame);
    };
}
```

```
    };
}
```

b. Now the version with rotation and beside:

```
function stack(painter1, painter2) {
    return rotate270(beside(rotate90(painter1),
                           rotate90(painter2)));
}
```

Answer of exercise 2.52

```
// Click here to play with any abstraction
// used for square_limit
```

Answer of exercise 2.53

```
list("a", "b", "c");
// ["a", ["b", ["c", null]]]

list(list("george"));
// [{"george", null}, null]

tail(list(list("x1", "x2"), list("y1", "y2")));
// [{"y1", ["y2", null]}, null]

tail(head(list(list("x1", "x2"), list("y1", "y2"))));
// ["x2", null]

memq("red", list(list("red", "shoes"), list("blue", "socks")));
// false

memq("red", list("red", "shoes", "blue", "socks"));
// [{"red", ["shoes", ["blue", ["socks", null]]]}]
```

Answer of exercise 2.54

```
function is_equal(a, b) {
    return (is_pair(a) && is_pair(b) &&
           is_equal(head(a), head(b)) && is_equal(tail(a), tail(b)))
    ||
    a === b;
}
```

Answer of exercise 2.55

The given expression consists of a single string that contains four single quotation marks.

Answer of exercise 2.56

```
function base(e) {
    return head(tail(e));
}
function exponent(e) {
    return head(tail(tail(e)));
}
function make_exp(base, exp) {
    return is_number_equal(exp, 0)
        ? 1
        : is_number_equal(exp, 1)
            ? base
            : list("**", base, exp);
}
function is_exp(x) {
    return is_pair(x) && head(x) === "**";
}
function deriv(exp, variable) {
    return is_number(exp)
        ? 0
        : is_variable(exp)
            ? (is_same_variable(exp, variable) ? 1 : 0)
            : is_sum(exp)
                ? make_sum(deriv(addend(exp), variable),
                           deriv(augend(exp), variable))
                : is_product(exp)
                    ? make_sum(make_product(multiplier(exp),
                                              deriv(multiplicand(exp),
                                                    variable)),
                               make_product(deriv(multiplier(exp),
                                                 variable),
                                             multiplicand(exp)))
                    : is_exp(exp)
                        ? make_product(make_product(exponent(exp),
                                                   make_exp(
                                                       base(exp),
                                                       exponent(exp) - 1)),
                                      deriv(base(exp), variable))
                        : Error("unknown expression type in deriv",
                               exp);
}
```

Answer of exercise 2.57

```
deriv(list("*", "x", "y", list("+", "x", 3)), "x");
```

Answer of exercise 2.58

a.

```
function make_sum(a1, a2) {
    return is_number_equal(a1, 0)
        ? a2
        : is_number_equal(a2, 0)
            ? a1
            : is_number(a1) && is_number(a2)
                ? a1 + a2
                : list(a1, "+", a2);
}
function is_sum(x) {
    return is_pair(x) && head(tail(x)) === "+";
}
function addend(s) {
    return head(s);
}
function augend(s) {
    return head(tail(tail(s)));
}
function make_product(m1, m2) {
    return is_number_equal(m1, 0) || is_number_equal(m2, 0)
        ? 0
        : is_number_equal(m1, 1)
            ? m2
            : is_number_equal(m2, 1)
                ? m1
                : is_number(m1) && is_number(m2)
                    ? m1 * m2
                    : list(m1, "*", m2);
}
function is_product(x) {
    return is_pair(x) && head(tail(x)) === "*";
}
function multiplier(s) {
    return head(s);
}
function multiplicand(s) {
    return head(tail(tail(s)));
}
function deriv(exp, variable) {
    return is_number(exp)
        ? 0
        : is_variable(exp)
            ? (is_same_variable(exp, variable) ? 1 : 0)
            : is_sum(exp)
                ? make_sum(deriv(addend(exp)), variable),
```

```

                deriv(augend(exp), variable))
: is_product(exp)
? make_sum(make_product(multiplier(exp),
                        deriv(multiplicand(exp),
                               variable)),
           make_product(deriv(multiplier(
                           exp),
                           variable),
                         multiplicand(exp)))
: Error("unknown expression type in deriv",
       exp);
}

```

b.

```

function items_before_first(op, s) {
    return head(s) === op
        ? null
        : pair(head(s),
               items_before_first(op, tail(s)));
}

function items_after_first(op, s) {
    return head(s) === op
        ? tail(s)
        : items_after_first(op, tail(s));
}

function make_sum(a1, a2) {
    return is_number_equal(a1, 0)
        ? a2
        : is_number_equal(a2, 0)
            ? a1
            : is_number(a1) && is_number(a2)
                ? a1 + a2
                : list(a1, "+", a2);
}

// a sequence of terms and operators is a sum
// if and only if at least one + operator occurs
function is_sum(x) {
    return is_pair(x) &&
        ! (is_null(member("+", x)));
}

function addend(s) {
    return items_before_first("+", s);
}

function augend(s) {
    return items_after_first("+", s);
}

function make_product(m1, m2) {
    return is_number_equal(m1, 0) || is_number_equal(m2, 0)
}

```

```

? 0
: is_number_equal(m1, 1)
? m2
: is_number_equal(m2, 1)
? m1
: is_number(m1) && is_number(m2)
? m1 * m2
: list(m1, "*", m2);
}
// a sequence of terms and operators is a product
// if and only if no + operator occurs
function is_product(x) {
    return is_pair(x) && is_null(member("+", x));
}
function multiplier(s) {
    return items_before_first("*", s);
}
function multiplicand(s) {
    return items_after_first("*", s);
}
function deriv(exp, variable) {
    return is_number(exp)
    ? 0
    : is_variable(exp)
    ? (is_same_variable(exp, variable) ? 1 : 0)
    : is_sum(exp)
    ? make_sum(deriv(addend(exp), variable),
               deriv(augend(exp), variable))
    : is_product(exp)
    ? make_sum(make_product(multiplier(exp),
                            deriv(multiplicand(exp),
                                  variable)),
               make_product(deriv(multiplier(exp),
                                 variable),
                            multiplicand(exp)))
    : Error("unknown expression type in deriv",
           exp);
}

```

Answer of exercise 2.59

```

function union_set(set1, set2) {
    return is_null(set1)
    ? set2
    : adjoin_set(head(set1),
                 union_set(tail(set1), set2));
}

```

Answer of exercise 2.60

The functions `is_element_of_set` and `intersection_set` remain unchanged. Here is the new implementation of `adjoin_set` and `union_set`.

```
function adjoin_set(x, set) {
    return pair(x, set);
}
function union_set(set1, set2) {
    return append(set1, set2);
}
```

In the version with no duplicates, the required number of steps for `is_element_of_set` and `adjoin_set` has an order of growth of $O(n)$, where n is the number of element occurrences in the given representation, and the required number of steps for `intersection_set` and `union_set` has an order of growth of $O(nm)$, where n is the number of element occurrences in the representation of the first set and m is the number of element occurrences in the representation of the second set. In the version that allows duplicates, the number of steps for `adjoin_set` shrinks to $O(n)$, and the number of steps for `union_set` shrinks to $O(n)$. However, note that the number of element occurrences may be much larger in the second version, because many duplicates may accumulate. For applications where duplicate elements are rare, the version that allows duplicates is preferable.

Answer of exercise 2.61

```
function adjoin_set(x, set) {
    return is_null(set)
        ? list(x)
        : x === head(set)
            ? set
            : x < head(set)
                ? pair(x, set)
                : pair(head(set),
                    adjoin_set(x, tail(set))));
}
```

Answer of exercise 2.62

```
function union_set(set1, set2) {
    if (is_null(set1)) {
        return set2;
    } else if (is_null(set2)) {
        return set1;
    } else {
        const x1 = head(set1);
        const x2 = head(set2);
```

```

    return x1 === x2
        ? pair(x1, union_set(tail(set1),
                              tail(set2)))
        : x1 < x2
        ? pair(x1, union_set(tail(set1), set2))
        : pair(x2, union_set(set1, tail(set2)));
    }
}

union_set(
    adjoin_set(10, adjoin_set(20, adjoin_set(30, null))),
    adjoin_set(10, adjoin_set(15, adjoin_set(20, null))));
```

Answer of exercise 2.63

- The two procedures produce the same results. For the trees in Figure 2.16, the result will always be `list(1, 3, 5, 7, 9, 11)`.
- A balanced tree with n elements has a height of $O(\log n)$ and $O(n)$ nodes. To convert the tree into a list using function `tree_to_list_1`, we call `tree_to_list_1` $O(n)$ times. We call `append` at each node of the tree, but at each level, we apply `append` with a combined $O(n)$ elements in the first arguments. Thus, the runtime of `tree_to_list_1` has an order of growth of $O(n \log n)$. Instead of `append`, the function `tree_to_list_2` gets away with calling `pair` at each node, and thus `tree_to_list_2` has an order of growth of $O(n)$.

Answer of exercise 2.64

- The function `partial_tree(elts, n)` returns a pair whose head is a balanced tree for the first $\lfloor (n - 1)/2 \rfloor$ elements of `elts`, and whose tail is the list containing the remaining elements of `elts`. It works by calling itself recursively, to construct the left subtree and right subtree, and then makes the tree, and the required return pair. Thus, the overall function `list_to_tree` just needs to call `partial_tree` with the given list and its length, and return the head of the result.

The tree for `list(1, 3, 5, 7, 9, 11)` is the tree on the right in Figure 2.16.

- The order of growth for the runtime of function `list_to_tree` is $O(n)$ because for every node of the result tree, only a constant amount of work is needed.

Answer of exercise 2.65

```
function union_set_as_tree(set1, set2) {
    const list1 = tree_to_list_2(set1);
    const list2 = tree_to_list_2(set2);
    return list_to_tree(union_set(list1, list2));
}

function intersection_set_as_tree(set1, set2) {
    const list1=tree_to_list_2(set1);
    const list2=tree_to_list_2(set2);
    return list_to_tree(intersection_set(list1, list2));
}
```

Answer of exercise 2.66

```
function lookup(given_key, tree_of_records) {
    if (is_null(tree_of_records)) {
        return null;
    } else {
        const this_entry = entry(tree_of_records);
        const this_key = key(this_entry);
        return given_key === this_key
            ? this_entry
            : given_key < this_key
                ? lookup(given_key,
                    left_branch(tree_of_records))
                : lookup(given_key,
                    right_branch(tree_of_records));
    }
}
```

Answer of exercise 2.67

```
decode(sample_message, sample_tree);
// should be: ["A", ["D", ["A", ["B", ["B", ["C", ["A", null]]]]]]]
```

Answer of exercise 2.68

```
function encode_symbol(symbol, tree) {
    function contains_symbol(symbol, current_tree) {
        return member(symbol, symbols(current_tree)) !== null;
    }
    if (is_leaf(tree)) {
        return null;
    } else {
        const left_tree = left_branch(tree);
        const right_tree = right_branch(tree);
```

```

    return contains_symbol(symbol, left_tree)
        ? pair(0, encode_symbol(symbol, left_tree))
        : contains_symbol(symbol, right_tree)
            ? pair(1, encode_symbol(symbol, right_tree))
            : error("symbol not found");
}
}

```

Answer of exercise 2.69

```

function successive_merge(leaves) {
    return length(leaves) === 1
        ? head(leaves)
        : successive_merge(
            adjoin_set(
                make_code_tree(head(leaves),
                    head(tail(leaves))),
                tail(tail(leaves))));
}

```

Answer of exercise 2.70

```

const lyrics_frequencies =
    list(list("A", 2),
        list("NA", 16),
        list("BOOM", 1),
        list("SHA", 3),
        list("GET", 2),
        list("YIP", 9),
        list("JOB", 2),
        list("WAH", 2));
const lyrics_tree = generate_huffman_tree(lyrics_frequencies);
const lyrics = list(
    'GET', 'A', 'JOB',
    'SHA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA',
    'GET', 'A', 'JOB', 'SHA', 'NA', 'NA', 'NA', 'NA', 'NA',
    'NA', 'NA', 'NA', 'WAH', 'YIP', 'YIP', 'YIP', 'YIP',
    'YIP', 'YIP', 'YIP', 'YIP', 'YIP', 'SHA', 'BOOM'
);
length(encode(lyrics, lyrics_tree));
// 84

```

We have an alphabet of $n = 8$ symbols, and a message of $m = 36$ symbols. Then the minimum number of bits to encode a specific symbol using a fixed-length code is $\lceil \log_2 n \rceil = 3$. Thus the minimum number of bits to encode all the lyrics is $m \lceil \log_2 n \rceil = 36 \times 3 = 108$.

Answer of exercise 2.71

The tree will be unbalanced, similar to the tree given in figure 2.17. Encoding the most frequent symbol requires one bit, whereas $n - 1$ bits are required to encode the least frequent symbol.

Answer of exercise 2.72

Consider the special case in exercise 2.68. At each step down the path of length n , we need to do a linear search in a list of length $n, n - 1, \dots, 1$. In the worst case, there are $O(n \times n/2) = O(n^2)$ number of steps. $O(n^2)$.

Answer of exercise 2.73

- Explain what was done above. Why can't we assimilate the predicates is_number and is_same_variable into the data-directed dispatch?*

The operator symbols come very handy as ‘type’ keys in the operator table. For numbers and variables, there aren’t such obvious keys, although we could introduce names for those types of expressions, as well, if we change the way expressions are represented as lists.

- Write the functions for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.*

```
function deriv_sum(operands, variable) {
    return make_sum(deriv(addend(operands), variable),
                    deriv(augend(operands), variable));
}
function deriv_product(operands, variable) {
    return make_sum(make_product(multiplier(operands),
                                 deriv(multiplicand(operands),
                                       variable)),
                    make_product(deriv(multiplier(
                                      operands),
                                      variable),
                                 multiplicand(operands)));
}
function install_deriv() {
    put("deriv", "+", deriv_sum);
    put("deriv", "*", deriv_product);
    return "done";
}
install_deriv();
```

- Choose any additional differentiation rule that you like, such as the one for exponents (Exercise 2.56), and install it in this data-directed system.*

```

function deriv_exponentiation(operands, variable) {
    const bas = base(operands);
    const exp = exponent(operands);
    return make_product(exp,
                        make_product(make_exponentiation(bas, make_sum(exp, -1)),
                                    deriv(bas, variable)));
}

function install_exponentiation_extension() {
    put("deriv", "**", deriv_exponentiation);
    return "done";
}
install_exponentiation_extension();

```

- d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the functions in the opposite way, so that the dispatch line in deriv looked like

```
get(operator(exp), "deriv")(operands(exp), variable);
```

What corresponding changes to the derivative system are required?

We would need to change the order of arguments in the installation procedure for the differentiation library:

```
put("+", "deriv", deriv_sum);
put("*", "deriv", deriv_product);
put("**", "deriv", deriv_exponentiation);
```

Answer of exercise 2.74

- a. Implement for headquarters a get_record function that retrieves a specified employee's record from a specified personnel file. The function should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?

We are tagging each division's file with a unique identifier for the division, using the tagging functions in section 2.4.2. We assume that each division provides an implementation of the get_record function and installs it in the company-wide operations table.

```

function make_insatiable_file(division, file) {
    return pair(division, file);
}
function insatiable_file_division(insatiable_file) {
    return head(insatiable_file);
}
function insatiable_file_content(insatiable_file) {
```

```

    return tail(insatiable_file);
}
function get_record(employee_name, insatiable_file) {
  const the_division
    = insatiable_file_division(insatiable_file);
  const division_record = get("get_record", the_division)
    (employee_name,
     insatiable_file_content(
       insatiable_file));
  return record !== undefined
    ? attach_tag(the_division, division_record)
    : undefined;
}

```

- b. *Implement for headquarters a get_salary function that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?*

Every division needs to implement functions such as get_salary and install them in Insatiable's operations table. Then, Insatiable's function get_salary can look like this:

```

function make_insatiable_record(division, record) {
  return pair(division, record);
}
function insatiable_record_division(insatiable_record) {
  return head(insatiable_record);
}
function insatiable_record_content(insatiable_record) {
  return tail(insatiable_record);
}
function get_salary(insatiable_record) {
  const the_division =
    insatiable_record_division(insatiable_record);
  return get("get_salary", the_division)
    (insatiable_record_content);
}

```

Note that we rely on the fact that any employee record that gets returned by get_record is tagged with its division, which is used in the generic function get_salary to retrieve the correct implementation from the operation table.

- c. *Implement for headquarters a find_employee_record function. This should search all the divisions' files for the record of a given employee and return the record. Assume that this function takes as arguments an employee's name and a list of all the divisions' files.*

```

function find_employee_record(employee_name,
                           personnel_files) {
  if (is_null(personell_files)) {

```

```

        return undefined;
    } else {
        const insatiable_record
            = get_record(employee_name,
                         head(personell_files));
        return insatiable_record !== undefined
            ? insatiable_record
            : find_employee_record(employee_name,
                                   tail(personell_files));
    }
}

```

- d. *When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?*

We would need to do the following, for each newly acquired company:

- Decide a name to be used as tag for any data item related to the new division.
- Write all division-specific functions such as `get_salary` and install them in the company-wide operations table using the division tag.
- Add the employee files to the list of `personell_files`. Note that this is a ‘destructive’ operation—similar to the extension of operations tables—in that the data structure is permanently and irrevocably modified; section 3.3 explains this concept in detail.

Answer of exercise 2.75

```

function make_from_mag_ang(r, a) {
    function dispatch(op) {
        return op === "real_part"
            ? r * math_cos(a)
            : op === "imag_part"
            ? r * math_sin(a)
            : op === "magnitude"
            ? r
            : op === "angle"
            ? a
            : Error("Unknown op in make_from_real_imag",
                    op);
    }
    return dispatch;
}

```

Answer of exercise 2.76

- *Generic operations with explicit dispatch*: For every new type, we need to touch every generic interface function, and add a new case.
- *Data-directed style*: Here the implementation of the generic interface functions can be neatly packaged in ‘install’ libraries for each new type. We can also have ‘install’ libraries for new operations.
- *Message-passing-style*: Like in the data-directed style, we need to write a library for each new type. In this case, the library consists of a dispatch function with a case for every generic interface function.

Overall, it’s probably best to use a data-directed style when we need to frequently add new operations, and message-passing, when we frequently add new types.

Answer of exercise 3.1

```
function make_accumulator(current) {
  function add(arg) {
    current = current + arg;
    return current;
  }
  return add;
}
```

Answer of exercise 3.2

```
const s = make_monitored(math_sqrt);
s(100);
display(s("how many calls"));
s(5);
display(s("how many calls"));

function make_monitored(f) {
  let counter = 0; //initialized to 0
  function mf(cmd) {
    if (cmd === "how many calls") {
      return counter;
    } else if (cmd === "reset count") {
      counter = 0;
      return counter;
    } else {
      counter = counter + 1;
      return f(cmd);
    }
  }
  return mf;
}
```

Answer of exercise 3.3

```
function make_account(balance, p) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    function dispatch(m, q) {
        if (p === q) {
            if (m === "withdraw") {
                return withdraw;
            } else if (m === "deposit") {
                return deposit;
            } else {
                return "Unknown request - make_account";
            }
        } else {
            return q => "Incorrect Password";
        }
    }
    return dispatch;
}

const a = make_account(100, "eva");
(a("withdraw", "eva"))(50); //withdraws 50
(a("withdraw", "ben"))(40); //incorrect password
```

Answer of exercise 3.4

```
function make_account(balance, p) {

    let invalid_attempts = 0; //initializes to 0

    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
}
```

```

}

function deposit(amount) {
    balance = balance + amount;
    return balance;
}

function call_the_cops() {
    return "calling the cops as you have exceeded " +
        "the max no of failed attempts";
}

function dispatch(m, q) {
    if (invalid_attempts <= 7) {
        if (p === q) {
            if (m === "withdraw") {
                return withdraw;
            } else if (m === "deposit") {
                return deposit;
            } else {
                return "Unknown request - make_account";
            }
        } else {
            invalid_attempts = invalid_attempts + 1;
            return "Incorrect Password";
        }
    } else {
        return call_the_cops();
    }
}

return dispatch;
}

```

References

- Abelson, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman, and Jack Wisdom. 1992. The Supercomputer Toolkit: A general framework for special-purpose computing. *International Journal of High-Speed Electronics* 3(3):337-361.
- Allen, John. 1978. *Anatomy of Lisp*. New York: McGraw-Hill.
- ANSI X3.226-1994. *American National Standard for Information Systems—Programming Language—Common Lisp*.
- Appel, Andrew W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25(4):275-279.
- Backus, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8):613-641.
- Baker, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4):280-293.
- Batali, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman, and Daniel Weise. 1982. The Scheme-81 architecture—System and chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, edited by Paul Penfield, Jr. Dedham, MA: Artech House.
- Borning, Alan. 1977. ThingLab—An object-oriented system for building simulations using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*.
- Borodin, Alan, and Ian Munro. 1975. *The Computational Complexity of Algebraic and Numeric Problems*. New York: American Elsevier.
- Chaitin, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5):47-52.
- Church, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton, N.J.: Princeton University Press.
- Clark, Keith L. 1978. Negation as failure. In *Logic and Data Bases*. New York: Plenum Press, pp. 293-322.
- Clinger, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings*

- of the ACM Symposium on Lisp and Functional Programming*, pp. 226-234.
- Clinger, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pp. 155-162.
- Colmerauer A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un système de communication homme-machine en franÇais. Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy.
- Cormen, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms*. Cambridge, MA: MIT Press.
- Darlington, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications*. New York: Cambridge University Press.
- Dijkstra, Edsger W. 1968a. The structure of the ‘THE’ multiprogramming system. *Communications of the ACM* 11(5):341-346.
- Dijkstra, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43-112.
- Dinesman, Howard P. 1968. *Superior Mathematical Puzzles*. New York: Simon and Schuster.
- deKleer, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pp. 116-125.
- Doyle, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12:231-272.
- Feigenbaum, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems*, vol. 9, pp. 105-117.
- Feeley, Marc. 1986. Deux approches à l'implantation du language Scheme. Masters thesis, Université de Montréal.
- Feeley, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1):47-66.
- Feller, William. 1957. *An Introduction to Probability Theory and Its Applications*, volume 1. New York: John Wiley & Sons.
- Fenichel, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11):611-612.
- Floyd, Robert. 1967. Nondeterministic algorithms. *JACM*, 14(4):636-644.
- Forbus, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers*. Cambridge, MA: MIT Press.

- Friedman, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium*, edited by S. Michaelson and R. Milner, pp. 257-284.
- Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages*. Cambridge, MA: MIT Press/McGraw-Hill.
- Gabriel, Richard P. 1988. The Why of Y. *Lisp Pointers* 2(2):15-25.
- Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley.
- Gordon, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh LCF*. Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag.
- Gray, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models*. San Mateo, CA: Morgan-Kaufman.
- Green, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 219-240.
- Green, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the ACM National Conference*, pp. 169-181.
- Griss, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah.
- Guttag, John V. 1977. Abstract data types and the development of data structures. *Communications of the ACM* 20(6):397-404.
- Hamming, Richard W. 1980. *Coding and Information Theory*. Englewood Cliffs, N.J.: Prentice-Hall.
- Hanson, Christopher P. 1990. Efficient stack allocation for tail-recursive languages. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pp. 106-118.
- Hanson, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers*, 4(3).
- Hardy, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2).
- Hardy, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers*. 4th edition. New York: Oxford University Press.
- Havender, J. 1968. Avoiding deadlocks in multi-tasking systems. *IBM Systems Journal* 7(2):74-84.
- Hearn, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University.

- Henderson, Peter. 1980. *Functional Programming: Application and Implementation*. Englewood Cliffs, N.J.: Prentice-Hall.
- Henderson, Peter. 1982. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 179-187.
- Hewitt, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 295-301.
- Hewitt, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3):323-364.
- Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1).
- Hodges, Andrew. 1983. *Alan Turing: The Enigma*. New York: Simon and Schuster.
- Hofstadter, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.
- Hughes, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17-42.
- IEEE Std 1178-1990. 1990. *IEEE Standard for the Scheme Programming Language*.
- Ingerman, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Unpublished manuscript. (Also, private communication from Wallace Feurzeig.)
- Kaldewaij, Anne. 1990. *Programming: The Derivation of Algorithms*. New York: Prentice-Hall.
- Kohlbecker, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University.
- Konopasek, Milos, and Sundaresan Jayaraman. 1984. *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education*. Berkeley, CA: Osborne/McGraw-Hill.
- Knuth, Donald E. 1973. *Fundamental Algorithms*. Volume 1 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley.
- Knuth, Donald E. 1981. *Seminumerical Algorithms*. Volume 2 of *The Art of Computer Programming*. 2nd edition. Reading, MA: Addison-Wesley.
- Kowalski, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh.
- Kowalski, Robert. 1979. *Logic for Problem Solving*. New York: North-Holland.
- Lamport, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558-565.

- Lampson, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek. 1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto.
- Landin, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89-101.
- Lieberman, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6):419-429.
- Liskov, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* 1(1):7-19.
- McAllester, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory.
- McAllester, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory.
- McCarthy, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4):184-195.
- McCarthy, John. 1967. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland.
- McCarthy, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages*.
- McCarthy, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual*. 2nd edition. Cambridge, MA: MIT Press.
- McDermott, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory.
- Miller, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3):300-317.
- Miller, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory.
- Moon, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science.
- Moon, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory.
- Morris, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages*.

- Phillips, Hubert. 1934. *The Sphinx Problem Book*. London: Faber and Faber.
- Pitman, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science.
- Rabin, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12:128-138.
- Raymond, Eric. 1993. *The New Hacker's Dictionary*. 2nd edition. Cambridge, MA: MIT Press.
- Raynal, Michel. 1986. *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press.
- Rees, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122.
- Rees, Jonathan, and William Clinger (eds). 1991. The revised⁴ report on the algorithmic language Scheme. *Lisp Pointers*, 4(3).
- Rivest, Ronald, Adi Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LCS/TM82, MIT Laboratory for Computer Science.
- Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1):23.
- Robinson, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1:107-124.
- Sagade, Y. 2015. [SICP exercise 1.14](#)
- Spafford, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the ACM* 32(6):678-688.
- Steele, Guy Lewis, Jr. 1977. Debunking the ‘expensive procedure call’ myth. In *Proceedings of the National Conference of the ACM*, pp. 153-62.
- Steele, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp. 98-107.
- Steele, Guy Lewis, Jr. 1990. *Common Lisp: The Language*. 2nd edition. Digital Press.
- Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory.
- Steele, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker's Dictionary*. New York: Harper & Row.
- Stoy, Joseph E. 1977. *Denotational Semantics*. Cambridge, MA: MIT Press.
- Sussman, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided

- circuit analysis. *IEEE Transactions on Circuits and Systems* CAS-22(11):857-865.
- Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierarchical descriptions. *AI Journal* 14:1-39.
- Sussman, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257:256-262.
- Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory.
- Sutherland, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory.
- Teitelman, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center.
- Thatcher, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 119-132. Turner, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics*, Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334-348.
- Wand, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):164-180.
- Waters, Richard C. 1979. A method for analyzing loop programs. *IEEE Transactions on Software Engineering* 5(3):237-247.
- Winograd, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory.
- Winston, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley.
- Zabih, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. *AAAI-87*, pp. 59-64.
- Zippel, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT.
- Zippel, Richard. 1993. *Effective Polynomial Computation*. Boston, MA: Kluwer Academic Publishers.

Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

— Donald E. Knuth, Fundamental Algorithms (Volume 1 of The Art of Computer Programming)

Page numbers for code definitions are in italics.

Page numbers followed by *n* indicate footnotes.

- absolute value, 30
- abstract data, 94
- abstract models for data, 102
- abstract syntax
 - in metacircular evaluator, 374
- abstraction
 - common pattern and, 69
 - functional, 38
 - metalinguistic, 371
- abstraction barriers, 93, 99–101, 182
 - in complex-number system, 183
 - in generic arithmetic system, 201
- accumulator, 127, 236
- Áchárya, Bhászcara, 53
- Ackermann’s function, 48
- acquire a mutex, 325
- Ada
 - recursive functions, 47
- Adams, Norman I., IV, 403
- adder
- full, 288
- half, 287
- ripple-carry, 291
- additivity, 94, 182, 192–199, 204
- Adelman, Leonard, 64
- A’h-mose, 58
- algebraic expression, 216
 - differentiating, 157–163
 - representing, 159–163
 - simplifying, 161–162
- algebraic specification for data, 102
- Algol
 - block structure, 42
 - call-by-name argument passing, 336, 412
 - thunks, 336, 411
- weakness in handling compound objects, 309
- algorithm
 - optimal, 132

probabilistic, 64, 229
aliasing, 245
alternative of `if`, 30
analyzing evaluator, 403–408
analyzing evaluator
 conditional expressions, 407
and-gate, 286
APL, 130
applicative-order evaluation, 29
 normal order vs., 33, 61, 409–411
applicative-order evaluation
 in JavaScript, 29
arbiter, 327
arctangent, 186
argument(s), 21
 delayed, 360
Aristotle’s *De caelo* (Buridan’s commentary on), 327
arithmetic
 generic, 201
 on complex numbers, 183
 on intervals, 105–109
 on power series, 346, 347
 on rational numbers, 95–99
ASCII code, 173
assignment, 231–248
 benefits of, 238–242
 bugs associated with, 245, 247
 costs of, 242–248
assignment operator, 232
atomic operations supported in hardware, 327
atomic requirement for test_and_set, 326
average damping, 82

B-tree, 170
Backus, John, 368
balanced binary tree, 170
balanced mobile, 123

bank account, 232, 263
joint, 245, 247
joint, with concurrent access, 311
password-protected, 237
stream model, 366
transferring money, 324
barrier synchronization, 329
Barth, John, 370
Basic
 restrictions on compound data, 110
 weakness in handling compound objects, 309
Bertrand’s Hypothesis, 343
binary search, 168
binary tree, 168
 balanced, 170
 converting a list to a, 171
 converting to a list, 170
 for Huffman encoding, 175
 represented with lists, 168
 sets represented as, 168–172
 table structured as, 285
bind, 40
binding, 248
 deep, 391
binomial coefficients, 53
black box, 38
block structure, 18, 41–42, 398–400
 in environment model, 261–264
blocked process, 326
body of a function, 25
Borning, Alan, 299
Borodin, Alan, 132
bound variable, 40
box-and-pointer notation, 109
 end-of-list marker, 111
branch
 branch of a tree, 23
browser, 19

bug, 17
 capturing a free name, 40
 order of assignments, 247
 side effect with aliasing, 245
 Buridan, Jean, 327
 busy-waiting, 326

C, 18
 recursive functions, 47
 restrictions on compound data, 110
 cache-coherence protocols, 312
 calculator, fixed points with, 81
 call-by-name argument passing, 336, 412
 call-by-need argument passing, 336, 412
 memoization and, 345
 canonical form, for polynomials, 224
 capturing a free name, 40
 Carmichael numbers, 64, 67
 case analysis
 data-directed programming vs., 376
 cell, in serializer implementation, 326
 Cesàro, Ernesto, 239
 Chaitin, Gregory, 238
 Chandah-sutra, 57
 change and sameness
 meaning of, 244–246
 shared data and, 269
 character, ASCII encoding, 173
 Chebyshev, Pafnutii L'vovich, 343
 chess, eight-queens puzzle, 138
 Chu Shih-chieh, 53
 Church numerals, 104
 Church, Alonzo, 75, 104
 Church-Turing thesis, 397
 circuit
 modeled with streams, 356, 361
 Clinger, William, 412
 closure, 93
 in abstract algebra, 110

 closure property of pair, 110
 closure property of picture-language operations, 140, 141
 lack of in many languages, 110
 code
 ASCII, 173
 fixed-length, 174
 Morse, 174
 prefix, 174
 variable-length, 174
 coercion, 208–215
 function, 209
 in algebraic manipulation, 225
 in polynomial arithmetic, 220
 table, 209
 combination, 20, 21
 combination as operator of, 85
 compound expression as operator of, 33
 evaluation of, 23
 lambda expression as operator of, 75
 as operator of combination, 85
 combination, means of, 19
 comments in programs, 136
 complex numbers
 polar representation, 187
 rectangular representation, 186
 rectangular vs. polar form, 184
 represented as tagged data, 188–192
 complex-number arithmetic, 183
 interfaced to generic arithmetic system, 204
 composition of functions, 89
 compound data, need for, 91–93
 compound expression, 20
 as operator of combination, 33
 compound function, 25
 compound procedure
 used like primitive procedure, 26

computability, 397, 398
computer science, 372, 397
 mathematics vs., 35
concrete data representation, 94
concurrency, 310–330
 correctness of concurrent programs,
 313
 deadlock, 328
 functional programming and, 368
 mechanisms for controlling, 316–330
concurrency
 correctness of concurrent programs,
 316
conditional statements, 77
congruent modulo n , 62
connector(s), in constraint system, 299
 operations on, 302
 representing, 305
consciousness, expansion of, 378
consequent
 of **if**, 30
constraint network, 299
constraint(s)
 primitive, 299
 propagation of, 298–309
constructor, 94
 as abstraction barrier, 99
continued fraction, 83
 e as, 84
 golden ratio as, 83
 tangent as, 84
conventional interface, 93
 sequence as, 125–139
Cormen, Thomas H., 170
correctness of a program, 35
cosine
 fixed point of, 81
 power series for, 345
cosmic radiation, 64
counting change, 51–53, 114
credit-card accounts, international, 329
cross-type operations, 208
cryptography, 64
cube root
 as fixed point, 85
 by Newton's method, 37
current time, for simulation agenda, 296
cycle in list, 268
 detecting, 272
Darlington, John, 368
data, 20
 abstract, 94
 abstract models for, 102
 algebraic specification for, 102
 compound, 91–93
 concrete representation of, 94
 functional representation of, 102–104
 hierarchical, 110, 118–123
 list-structured, 97
 meaning of, 102–104
 numerical, 20
 as program, 395–397
 shared, 269–272
 symbolic, 154
 tagged, 188–192
data abstraction, 92, 94, 182, 185, 381
 for queue, 274
data base
 data-directed programming and, 198
 Insatiable Enterprises personnel, 198
 as set of records, 172
data types
 in Lisp, 207
 in strongly typed languages, 364
data-directed programming, 183, 192–199
 case analysis vs., 376
 in metacircular evaluator, 385

data-directed recursion, 220
deadlock, 328
 avoidance, 328
 recovery, 328
declarative vs. imperative knowledge, 35
decomposition of program into parts, 38
deep binding, 391
deferred operations, 46
definite integral, 71–72
 estimated with Monte Carlo simulation, 241, 366
delay, in digital circuit, 286
delayed argument, 360
delayed evaluation, 231, 330
 assignment and, 338
 explicit vs. automatic, 420
 in lazy evaluator, 409–421
 normal-order evaluation and, 363–364
 printing and, 337
 streams and, 358–363
dense polynomial, 221
deque, 279
derivative of a function, 86
design, stratified, 153
differential equation, 359
 second-order, 361
differentiation
 numerical, 86
 rules for, 157, 162
 symbolic, 157–163, 197
diffusion, simulation of, 316
digital signal, 286
digital-circuit simulation, 286–298
 agenda, 293–294
 agenda implementation, 295–298
 primitive function boxes, 289–291
 representing wires, 291–293
 sample simulation, 294–295
Dijkstra, Edsger Wybe, 325
Diophantus’s *Arithmetic*, Fermat’s copy of, 62
dispatching
 comparing different styles, 200
 on type, 192
dog, perfectly rational, behavior of, 327
driver loop
 in lazy evaluator, 413
 in metacircular evaluator, 394
dynamic typing, 18, 19

e
 as continued fraction, 84
 as solution to differential equation, 360
 e^x , power series for, 345
Earth, measuring circumference of, 339
efficiency
 of tree-recursive process, 53
efficiency
 of evaluation, 403
Eich, Brendan, 18
EIEIO, 329
eight-queens puzzle, 138
electrical circuits, modeled with streams, 356, 361
embedded language, language design
 using, 409
empty list, 112
 recognizing with `null?`, 112
encapsulated name, 234
enclosing environment, 248
end-of-list marker, 111
engineering vs. mathematics, 64
enumerator, 127
environment, 23, 248
 as context for evaluation, 24
 enclosing, 248
 lexical scoping and, 42
environment model of evaluation, 231,

248–264
environment structure, 249
function-application example, 253–255
internal definitions, 261–264
local state, 255–260
message passing, 263
metacircular evaluator and, 373
rules for evaluation, 250–252
tail recursion and, 255

equality
in generic arithmetic system, 207
of lists, 156
of numbers, 30
referential transparency and, 244

Eratosthenes, 339

Escher, Maurits Cornelis, 140

Euclid’s Algorithm, 59–61
order of growth, 60
for polynomials, 226

Euclid’s *Elements*, 60

Euclid’s proof of infinite number of primes, 343

Euclidean ring, 226

Euler, Leonhard, 84
proof of Fermat’s Little Theorem, 62
series accelerator, 348

evaluation
of a combination, 23
of an operator combination, 24
of or, 31
of primitive expressions, 24

evaluator, 371
as abstract machine, 396
metacircular, 373
as universal machine, 396

event-driven simulation, 286

execution function
in analyzing evaluator, 403

exponential growth, 55
of tree-recursive Fibonacci-number computation, 50

exponentiation, 56–57
modulo n , 63

expression
self-evaluating, 375
symbolic, 93
as a tree, 23

expression-oriented vs. imperative programming style, 309

factorial, 44
infinite stream, 343
without letrec or define, 402

false, 30

feedback loop, modeled with streams, 358

Feeley, Marc, 403

Fermat, Pierre de, 62

Fermat test for primality, 62–64
variant of, 67

Fermat’s Little Theorem, 62
alternate form, 67
proof, 62

Fibonacci numbers, 49
Euclid’s GCD algorithm and, 60

FIFO buffer, 274

filter, 73, 127

first-class elements in language, 88

fixed point, 81–82
computing with calculator, 81
of cosine, 81
cube root as, 85
fourth root as, 90
golden ratio as, 82
as iterative improvement, 90
in Newton’s method, 85
nth root as, 90
square root as, 81, 85, 87
of transformed function, 87

fixed-length code, 174
 flatmap, 136
 force a thunk, 411
 formal parameters
 names of, 40
 scope of, 40
 Fortran, 130
 inventor of, 368
 restrictions on compound data, 110
 fourth root, as fixed point, 90
 frame (environment model), 248
 as repository of local state, 255–260
 global, 248
 frame (picture language), 140, 147
 coordinate map, 147
 free name
 capturing, 40
 free variable, 40
 in internal definition, 42
 Friedman, Daniel P., 336, 372
 full-adder, 288
 function
 as argument, 69–74
 body of, 25
 compound, 25
 creating with define, 25
 creating with **function**, 252
 creating with lambda, 250
 declaration of, 25, 26
 first-class in Lisp, 88
 as general method, 79
 generic, 178, 183
 memoized, 285
 monitored, 237
 name of, 25
 naming (with define), 25
 parameters of, 25
 as pattern for local evolution of a
 process, 43
 as returned value, 90
 special form vs., 410
 statically-scoped, first-class, 18, 19
 function
 as general method, 84
 special form vs., 419
 function (mathematical)
 \mapsto notation for, 81
 Ackermann's, 48
 composition of, 89
 derivative of, 86
 fixed point of, 81–82
 procedure vs., 34–35
 rational, 225–229
 repeated application of, 89
 smoothing of, 89
 function application
 environment model of, 253–255
 function box, in digital circuit, 286
 functional abstraction, 38
 functional programming, 242, 364–369
 concurrency and, 368
 functional programming languages,
 368
 time and, 366–369
 functional representation of data, 102–104
 Gabriel, Richard P., 402
 garbage collection
 memoization and, 415
 generic arithmetic operations, 202–207
 generic function, 178, 183
 generic selector, 190, 192
 generic operation, 94
 glitch, 17
 global environment, 23, 249
 in metacircular evaluator, 391
 global frame, 248
 Goguen, Joseph, 102

golden ratio, 50
 as continued fraction, 83
 as fixed point, 82
 Gordon, Michael, 364
 Gray, Jim, 328
 greatest common divisor, 59–61
 generic, 226
 of polynomials, 226
 used to estimate π , 239
 used in rational-number arithmetic, 98
 Guttag, John Vogel, 102

half-adder, 287
 simulation of, 294–295

half-interval method, 79–80
 half_interval_method, 80
 Newton's method vs., 86

halting problem, 398
 Halting Theorem, 398
 Hamming, Richard Wesley, 176, 343
 Hardy, Godfrey Harold, 343, 355
 Hassle, 410
 Havender, J., 328
 Haynes, Christopher T., 372
 headed list, 279, 296
 Henderson, Peter, 140, 340, 368
 Henderson diagram, 340
 Heraclitus, 230
 Heron of Alexandria, 35
 Hewitt, Carl Eddie, 47
 hiding principle, 234
 hierarchical data structures, 110, 118–123
 hierarchy of types, 211–215
 in symbolic algebra, 224–225
 inadequacy of, 213
 high-level language, machine language vs., 370
 higher-order functions
 function as general method, 84
 in metacircular evaluator, 377
 higher-order functions, 68
 function as argument, 69–74
 function as general method, 79
 function as returned value, 90
 higher-order functions
 strong typing and, 364
 Hilfinger, Paul, 172
 Hoare, Charles Antony Richard, 102
 Hodges, Andrew, 397
 Hofstadter, Douglas R., 397
 Horner, W. G., 131
 Horner's rule, 131
 Huffman code, 173–182
 optimality of, 176
 order of growth of encoding, 181
 Huffman, David, 175
 Hughes, R. J. M., 420

imperative programming, 246
 imperative vs. declarative knowledge, 35
 imperative vs. expression-oriented
 programming style, 309

indeterminate of a polynomial, 216
 infinite stream(s), 339–347
 merging, 344, 352, 354, 368
 merging as a relation, 369
 of factorials, 343
 of pairs, 351–355
 representing power series, 345
 to model signals, 355–358
 to sum a series, 348

infix notation, 21
 infix notation, prefix notation vs., 163
 Ingerman, Peter, 411
 integer vs. real number, 20
 integer(s), 20
 integerizing factor, 227
 integral

of a power series, 345
integrator, for signals, 355
internal declaration
 scope of name, 398
internal definition, 41–42
 in environment model, 261–264
free variable in, 42
position of, 42
restrictions on, 399
scanning out, 399
 scope of name, 400
Internet Explorer, 18
interpreter, 18
 read-eval-print loop, 22
interval arithmetic, 105–109
invariant quantity of an iterative process,
 58
inverter, 286
iterative improvement, 90
iterative process, 46
 as a stream process, 347–351
 design of algorithm, 58
 implemented by function call, 47
 implemented by procedure call, 36
 linear, 46, 54
 recursive process vs., 44–47, 255

Java, 18
JavaScript
 history of, 18
JavaScript
 applicative-order evaluation in, 29
Jayaraman, Sundaresan, 299
JScript, 19

Kaldewaij, Anne, 59
Karr, Alphonse, 230
key of a record
 in a data base, 172
 in a table, 279

testing equality of, 284
Khayyam, Omar, 53
Knuth, Donald E., 53, 57, 60, 131, 238, 239
Kolmogorov, A. N., 238
Konopasek, Milos, 299
KRC, 135, 353

Lagrange interpolation formula, 217
 λ calculus (lambda calculus), 75
Lambert, J.H., 84
Lamé, Gabriel, 60
Lamé’s Theorem, 60
Lamport, Leslie, 329
Lampson, Butler, 245
Landin, Peter, 336
Lapalme, Guy, 403
lazy evaluation, 409
lazy evaluator, 409–419
lazy list, 419–421
lazy pair, 419–421
lazy tree, 420
least commitment, principle of, 188
lecture, something to do during, 81
Leibniz, Baron Gottfried Wilhelm von
 proof of Fermat’s Little Theorem, 62
 series for π , 348
Leibniz, Baron Gottfried Wilhelm von
 series for π , 69
Leiserson, Charles E., 170, 355
lexical scoping, 42
line segment
 represented as pair of points, 101
 represented as pair of vectors, 150
linear growth, 46, 55
linear iterative process, 46
 order of growth, 54
linear recursive process, 46
 order of growth, 54
Liskov, Barbara Huberman, 102

Lisp, 18
 first-class functions in, 88
 internal type system, 207
 suitability for writing evaluators, 372

list structure, 97
 list vs., 111
 mutable, 265–269

list(s), 111
 tailing down, 112
 combining with append, 113
 pairing up, 113
 converting a binary tree to a, 170
 converting to a binary tree, 171
 equality of, 156
 headed, 279, 296
 last pair of, 113
 lazy, 419–421
 length of, 112
 list structure vs., 111
 manipulation with head, tail, and
 pair, 111
 mapping over, 116–118
 nth element of, 112
 operations on, 112
 reversing, 114
 techniques for manipulating, 112

LiveScript, 18

local evolution of a process, 43

local name, 39–40, 75

local state, 231–248
 maintained in frames, 255–260

local state variable, 232–238

local variable, 75

Locke, John, 17

logarithm, approximating $\ln 2$, 351

logarithmic growth, 55, 57, 168

logical and, 286

logical or, 287

looping constructs, 36, 47

machine language
 high-level language vs., 370

mapping
 over lists, 116–118
 nested, 135–139, 351–355
 as a transducer, 127
 over trees, 124–125

→ notation for mathematical function, 81

mathematics
 computer science vs., 35
 engineering vs., 64

matrix, represented as sequence, 133

means of abstraction, 19
 var, 22

means of combination, 19

measure in a Euclidean ring, 226

memoization, 53, 285
 call-by-need and, 345
 by delay, 336
 garbage collection and, 415
 of thunks, 411

message passing, 103, 199–201
 environment model and, 263
 in bank account, 236
 in digital-circuit simulation, 291
 tail recursion and, 47

metacircular evaluator, 373

metacircular evaluator for JavaScript
 primitive functions, 391
 undefined, 391

metacircular evaluator for Scheme,
 372–398
 data abstraction in, 373, 390
 driver loop, 394
 environment model of evaluation in,
 373
 environment operations, 387
 eval-apply cycle, 373
 expression representation, 381

global environment, 391
higher-order functions in, 377
job of, 373
primitive functions, 393
representation of environments,
 388–391
representation of functions, 386
representation of true and false, 386
running, 391–395
special forms (additional), 385
symbolic differentiation and, 381
syntax of evaluated language, 381
metacircular evaluator for Scheme
 analyzing version, 403–408
 data abstraction in, 374
 data-directed eval, 385
 efficiency of, 403
 eval and apply, 374–380
 expression representation, 374
 implemented language
 vs. implementation language, 378
 order of operand evaluation, 380
metalinguistic abstraction, 371
Microsoft, 18
Miller, Gary L., 67
Miller-Rabin test for primality, 67
Milner, Robin, 364
Miranda, 135
ML, 364
mobile, 123
Mocha, 18
modeling
 as a design strategy, 230
 in science and engineering, 28
modularity, 129, 230
 along object boundaries, 369
functional programs vs. objects,
 364–369
hiding principle, 234
streams and, 347
through dispatching on type, 192
through infinite streams, 366
through modeling with objects, 238
modulo n , 62
monitored function, 237
Monte Carlo integration, 241
 stream formulation, 366
Monte Carlo simulation, 239
 stream formulation, 364
Morris, J. H., 245
Morse code, 174
multiplication by Russian peasant method,
 58
Munro, Ian, 132
mutable data objects, 264–274
 implemented with assignment,
 272–274
 list structure, 265–269
 pairs, 265–269
 procedural representation of, 272–274
 shared data, 271
mutator, 264
mutex, 325
mutual exclusion, 325
name
 encapsulated, 234
 of a formal parameter, 40
 of a function, 25
naming
 of computational objects, 22
 of functions, 25
Netscape Communications Corporation, 18
Netscape Navigator, 18
Newton's method
 for cube roots, 37
 for differentiable functions, 85–87
 half-interval method vs., 86

for square roots, 35–36, 86, 87
 node of a tree, 23
 non-computable, 398
 non-strict, 410
 nondeterminism, in behavior of concurrent programs, 316, 369
 normal-order evaluation, 29
 applicative order vs., 33, 61, 409–411
 delayed evaluation and, 363–364
 of `if`, 34
 notation in this book
 italic symbols in expression syntax, 25
*n*th root, as fixed point, 90
 number theory, 62
 number(s), 20
 comparison of, 30
 equality of, 30
 in generic arithmetic system, 202
 numerical analysis, 20
 numerical analyst, 80
 numerical data, 20

 object(s), 231
 benefits of modeling with, 238
 with time-varying state, 232
 object-oriented programming languages, 213

 operands, 21

 operation
 cross-type, 208
 generic, 94
 operation-and-type table, 193
 assignment needed for, 232

 operator, 21

 operator combination
 evaluation of, 24

 operator of a combination
 combination as, 85
 compound expression as, 33

 operator of a combination
 lambda expression as, 75

 optimality
 of Horner's rule, 132
 of Huffman code, 176

 or-gate, 287
 `or_gate`, 290

 order notation, 54

 order of evaluation
 assignment and, 248
 in metacircular evaluator, 380
 in Scheme, 248

 order of events
 decoupling apparent from actual, 335

 order of events
 indeterminacy in concurrent systems, 311

 order of growth, 54–55
 linear iterative process, 54
 linear recursive process, 54
 logarithmic, 57
 tree-recursive process, 54

 ordered-list representation of sets, 166–167

 ordinary numbers (in generic arithmetic system), 202

 Ostrowski, A. M., 132

 P operation on semaphore, 325

 package, 194
 complex-number, 204
 polar representation, 195
 polynomial, 218
 rational-number, 203
 rectangular representation, 194
 Scheme-number, 203

 painter(s), 140
 higher-order operations, 145
 operations, 141
 represented as functions, 149

transforming and combining, 150
 pair(s), 96
 axiomatic definition of, 102
 box-and-pointer notation for, 109
 functional representation of, 104
 infinite stream of, 351–355
 lazy, 419–421
 mutable, 265–269
 procedural representation of, 102–274,
 419
 used to represent sequence, 111
 used to represent tree, 118–123
 Pan, V. Y., 132
 parameters, 25
 parentheses
 in function definition, 26
 Pascal
 lack of higher-order functions, 364
 recursive functions, 47
 restrictions on compound data, 110
 weakness in handling compound
 objects, 309
 Pascal, Blaise, 53
 Pascal’s triangle, 53
 password-protected bank account, 237
 Perlis, Alan J., 110
 permutations of a set, 136
 π (pi)
 approximation with half-interval
 method, 80
 approximation with Monte Carlo
 integration, 241, 366
 Cesàro estimate for, 239, 364
 Leibniz’s series for, 348
 stream of approximations, 348–350
 Wallis’s formula for, 72
 π (pi)
 Leibniz’s series for, 69
 picture language, 140–154
 Pingala, Áchárya, 57
 pipelining, 311
 point, represented as a pair, 101
 pointer
 in box-and-pointer notation, 109
 poly, 217
 polynomial arithmetic, 216–229
 addition, 217–221
 division, 223
 Euclid’s Algorithm, 226
 greatest common divisor, 226–229
 interfaced to generic arithmetic
 system, 218
 multiplication, 217–221
 probabilistic algorithm for GCD, 229
 rational functions, 225–229
 subtraction, 222
 polynomial(s), 216–229
 canonical form, 224
 dense, 221
 evaluating with Horner’s rule, 131
 hierarchy of types, 224–225
 indeterminate of, 216
 sparse, 221
 univariate, 216
 power series, as stream, 345
 adding, 346
 dividing, 347
 integrating, 345
 multiplying, 346
 PowerPC, 329
 predicate, 30
 of **if**, 30
 prefix code, 174
 prefix notation
 infix notation vs., 163
 prime number(s), 61–64
 cryptography and, 64
 Eratosthenes’s sieve for, 339

Fermat test for, 62–64
Miller-Rabin test for, 67
testing for, 61–67

primitive constraints, 299

primitive expression, 19
evaluation of, 24

principle of least commitment, 188

probabilistic algorithm, 64, 229, 340

procedural representation of data
mutable data, 272–274

procedure, 20
anonymous, 74
as black box, 38–39
mathematical function vs., 34–35
scope of formal parameters, 40

process, 17
iterative, 46
linear iterative, 46
linear recursive, 46
local evolution of, 43
order of growth of, 54
recursive, 46
resources required by, 54
shape of, 46
tree-recursive, 49–53

program
as abstract machine, 395
comments in, 136
as data, 395–397
structure of, 38, 40–42
structured with subroutines, 397

programming
demand-driven, 335
elements of, 19, 20
imperative, 246
odious style, 337

programming language, 17
design of, 409
functional, 368

object-oriented, 213
strongly typed, 364
very high-level, 35

prompts, 394

propagation of constraints, 298–309

proving programs correct, 35

pseudo-random sequence, 238

pseudodivision of polynomials, 227

pseudoremainder of polynomials, 227

puzzles
eight-queens puzzle, 138

Pythagorean triples
with streams, 354

quantum mechanics, 369

queue, 274–279
double-ended, 279
front of, 274
operations on, 274
procedural implementation of, 278
rear of, 274
in simulation agenda, 295

Rabin, Michael O., 67

radicand, 35

Ramanujan numbers, 355

Ramanujan, Srinivasa, 355

random-number generator, 232, 238
in Monte Carlo simulation, 239
in primality testing, 62
with reset, 242
with reset, stream version, 366

rational function, 225–229
reducing to lowest terms, 228–229

rational number(s)
arithmetic operations on, 95–99
printing, 97
reducing to lowest terms, 98, 100
represented as pairs, 97

rational-number arithmetic, 95–99

interfaced to generic arithmetic system, 203
need for compound data, 92
Raymond, Eric, 409
RC circuit, 356
read-eval-print loop, 22
real number, 20
record, in a data base, 172
rectangle, representing, 101
recursion, 23
 data-directed, 220
 expressing complicated process, 23
 in working with trees, 119
recursion theory, 397
recursive function
 recursive process vs., 46
recursive procedure
 recursive procedure definition, 38
 specifying without define, 402
recursive process, 46
 iterative process vs., 44–47, 255
 linear, 46, 54
 recursive function vs., 46
 tree, 49–54
red-black tree, 170
reducing to lowest terms, 98, 100, 228–229
Rees, Jonathan A., 403
referential transparency, 244
relations, computing in terms of, 299
relatively prime, 74
relativity, theory of, 329
release a mutex, 325
remainder modulo n , 62
resistance
 formula for parallel resistors, 105, 108
 tolerance of resistors, 105
Reuter, Andreas, 328
Rhind Papyrus, 58
ripple-carry adder, 291
Rivest, Ronald L., 64, 170
RLC circuit, 361
robustness, 153
rock songs, 1950s, 181
roundoff error, 20, 184
RSA algorithm, 64
Russian peasant method of multiplication, 58
sameness and change
 meaning of, 244–246
 shared data and, 269
scanning out internal definitions, 399
Scheme
 as precursor of JavaScript, 18
Schmidt, Eric, 245
scope of a name
 internal declaration, 398
scope of a variable, 40
 procedure's formal parameters, 40
search
 of binary tree, 168
selector, 94
 as abstraction barrier, 99
 generic, 190, 192
Self, 18
self-evaluating expression, 375
semaphore, 325
 of size n , 327
semicolon
 comment introduced by, 136
separator code, 174
sequence accelerator, 348
sequence of expressions
 in function body, 26
sequence(s), 111
 as conventional interface, 125–139
 as source of modularity, 129
 operations on, 128–135

represented by pairs, 111
serializer, 317–321
implementing, 325–327
with multiple shared resources,
321–325
series, summation of, 70
accelerating sequence of
approximations, 348
with streams, 348
set, 163
data base as, 172
operations on, 163–164
permutations of, 136
represented as binary tree, 168–172
represented as ordered list, 166–167
represented as unordered list, 164–165
subsets of, 125
shadow a binding, 249
Shamir, Adi, 64
shape of a process, 46
shared data, 269–272
shared resources, 321–325
shared state, 312
side-effect bug, 245
sieve of Eratosthenes, 339
signal processing
smoothing a function, 89
smoothing a signal, 358
stream model of, 355–358
zero crossings of a signal, 357, 358
signal, digital, 286
signal-flow diagram, 127
signal-processing view of computation, 127
simplification of algebraic expressions, 161
Simpson’s Rule for numerical integration,
71
simulation
event-driven, 286
sine
approximation for small angle, 55
power series for, 345
SKETCHPAD, 299
Smalltalk, 299
smoothing a function, 89
smoothing a signal, 358
snarf, 409
Solomonoff, Ray, 238
sparse polynomial, 221
special form
need for, 36
special form
function vs., 410, 419
square, 25
square root, 35–36
stream of approximations, 347
stack, 46
Stallman, Richard M., 299
state
shared, 312
vanishes in stream formulation, 367
state names, 46
state variable, 231
local, 232–238
Steele, Guy Lewis Jr., 47, 247, 299, 409
Stoy, Joseph E., 28, 59, 402
Strachey, Christopher, 88
stratified design, 153
stream(s), 231, 330–369
delayed evaluation and, 358–363
implemented as delayed lists, 331–333
implemented as lazy lists, 419–421
implicit definition, 341–343
strict, 410
strings, 154–157
strongly typed language, 364
substitution model of function application,
248
inadequacy of, 242–244

shape of process, 46
substitution model of procedure
 application, 27, 30
subtype, 211
 multiple, 213
successive squaring, 56
summation of a series, 70
 with streams, 348
Sun Microsystems, 18
supertype, 211
 multiple, 213
Sussman, Gerald Jay, 47, 299
Sussman, Julie Esther Mazel, nieces of, 154
Sutherland, Ivan, 299
symbol(s), 154
 quotation of, 155
symbolic algebra, 216–229
symbolic differentiation, 157–163, 197
symbolic expression, 93
SYNC, 329
syntactic analysis, separated from
 execution
 in metacircular evaluator, 403–408
syntactic sugar
 function vs. data as, 292
 define, 382
 looping constructs as, 47
syntax
 of a programming language, 24
syntax interface, 292

table, 279–286
 backbone of, 279
 for coercion, 209
 for data-directed programming, 193
 local, 283–284
 n-dimensional, 285

one-dimensional, 279–281
represented as binary tree
 vs. unordered list, 285
testing equality of keys, 284
two-dimensional, 281–283
used in simulation agenda, 296
used to store computed values, 285

tableau, 349

tabulation, 53, 285

tagged data, 188–192

tail recursion, 47
 environment model of evaluation, 255
 in Scheme, 47

tangent
 as continued fraction, 84
 power series for, 347

Technological University of Eindhoven, 325

term list of polynomial, 217
 representing, 221–223

terminal node of a tree, 23

Thatcher, James W., 102

THE Multiprogramming System, 325

$\theta(f(n))$ (theta of $f(n)$), 54

thunk, 411–412
 call-by-name, 336
 call-by-need, 336
 forcing, 411
 implementation of, 414–415
 origin of name, 411

time
 assignment and, 310
 communication and, 329
 in concurrent systems, 311
 functional programming and, 366–369
 purpose of, 311

time
 in concurrent systems, 316

time segment, in agenda, 295

time slicing, 327
TK
 Solver, 299
tower of types, 211
transparency, referential, 244
tree
 B-tree, 170
 binary, 168
 combination viewed as, 23
 counting leaves of, 119
 enumerating leaves of, 129
 fringe of, 122
 Huffman, 175
 lazy, 420
 mapping over, 124–125
 red-black, 170
 represented as pairs, 118–123
 reversing at all levels, 122
tree accumulation, 24
tree-recursive process, 49–53
 order of growth, 54
trigonometric relations, 187
true, 30
truncation error, 20
Turing machine, 397
Turing, Alan M., 397, 398
Turner, David, 135, 353, 368
type tag, 183, 188
 two-level, 206
type(s)
 cross-type operations, 208
 dispatching on, 192
 hierarchy in symbolic algebra, 224–225
 hierarchy of, 211–215
 lowering, 212, 215
 multiple subtype and supertype, 213
 raising, 211, 214
 subtype, 211
 supertype, 211
tower of, 211
type-inferencing mechanism, 364
typing
 dynamic, 18, 19
unbound variable, 248
unit square, 147
univariate polynomial, 216
universal machine, 396
unordered-list representation of sets, 164–165
unspecified values
 display, 97
 if without alternative, 297
 newline, 97
 set_head, 266
 set_tail, 266
upward compatibility, 418
V operation on semaphore, 325
variable
 bound, 40
 free, 40
 scope of, 40
 unbound, 248
 value of, 248
variable-length code, 174
vector (mathematical)
 operations on, 133, 148
 in picture-language frame, 147
 represented as pair, 148
 represented as sequence, 133
Venus, 155
very high-level language, 35
Wadler, Philip, 245
Wadsworth, Christopher, 364
Wagner, Eric G., 102
Wallis, John, 72
Wand, Mitchell, 372

- Waters, Richard C., 130
Weyl, Hermann, 91
width of an interval, 106
Wiles, Andrew, 62
wire, in digital circuit, 286
Wise, David S., 336
wishful thinking, 95, 158
world line of a particle, 330, 367
Wright, E. M., 343
Wright, Jesse B., 102
Xerox Palo Alto Research Center, 299
 Y operator, 402
zero crossings of a signal, 357, 358
zero test (generic), 207
 for polynomials, 222
Zilles, Stephen N., 102
Zippel, Richard E., 229

JavaScript Adaptation Making-of

Background

Like the Source Academy, the JavaScript adaptation of SICP is an open-source community effort. The software and data required for making these web pages and the PDF edition are contained in the repository [Source Academy / sicp](#), and improvements, extensions and discussions are handled in this repository as with many other open-source software projects.

Martin Henz started translating SICP to JavaScript in 2008. He obtained the original \LaTeX sources of the textbook from the Gerald Sussman, and converted it to an XML format that allowed him to retain the original sources along with the JavaScript adaptation in a single file. He developed a processing system to generate HTML from XML, using XSLT, resulting in the first version of the JavaScript adaptation.

Mobile-friendly Web Edition

The [mobile-friendly web edition](#) of SICP JS was designed and implemented by Liu Hang in 2017 and then further developed by Feng Piaopiao in 2018. Liu Hang decided to use Ruby on Rails for generating the HTML pages from the XML sources. The XML documents are processed using Nokogiri. For that, the website is originally hosted as a Ruby on Rails application and the generated HTML files are then collected as a pure-HTML5 website. Formulas are retained in the resulting HTML files and are type-set by the reader's browser on-the-fly, using the MathJax system.

In the textbook, program fragments often require other program fragments. In order to collect and execute the necessary programs, the corresponding SNIPPET tags in the xml files include REQUIRES tags. The Rails server uses these tags in order to assemble the executable programs.

PDF Edition

The [PDF edition](#) of SICP JS was designed and implemented by Chan Ger Hean in 2019. Ger Hean decided to use Node.js for generating files from the XML sources. The files are then typeset using the PdfLaTeX system.

E-book Edition

The [e-book edition](#) of SICP JS was designed and implemented by Jolyn Tan in 2019. Jolyn decided to use Node.js for generating files from the XML sources. The files are then processed into the EPUB 3 format using the pandoc system.

Figures

The figures are adapted from [HTML5/EPUB3 version of SICP](#) by Andres Raba. The figures are licensed under Creative Commons Attribution-ShareAlike 4.0 International License ([cc by-sa](#)). JavaScript adaptations of figures were done by manually by Tobias Wrigstad using Inkscape and gratuitous use of sed.