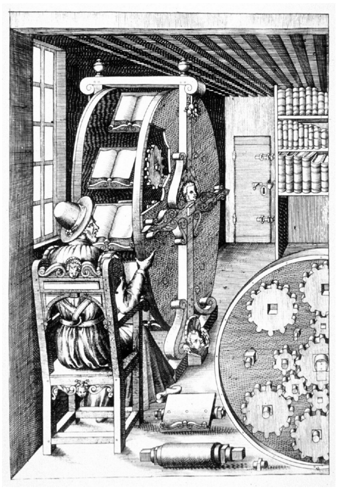
Structure and Interpretation of Computer Programs



SECOND EDITION

Unofficial Texinfo Format

2.andresraba5.3

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman,
foreword by Alan J. Perlis

Structure and Interpretation of Computer Programs,
second edition

Harold Abelson and Gerald Jay Sussman
with Julie Sussman, foreword by Alan J. Perlis

The MIT Press
Cambridge, Massachusetts
London, England

McGraw-Hill Book Company
New York, St. Louis, San Francisco,
Montreal, Toronto

Unofficial Texinfo Format 2.andresraba5.3 (April 6, 2014),
based on 2.neilvandyke4 (January 10, 2007).

Contents

Unofficial Texinfo Format

This is the second edition SICP book, from Unofficial Texinfo Format.

You are probably reading it in an Info hypertext browser, such as the Info mode of Emacs. You might alternatively be reading it TeX-formatted on your screen or printer, though that would be silly. And, if printed, expensive.

The freely-distributed official HTML-and-GIF format was first converted personally to Unofficial Texinfo Format (UTF) version 1 by Lytha Ayth during a long Emacs lovefest weekend in April, 2001.

The UTF is easier to search than the HTML format. It is also much more accessible to people running on modest computers, such as donated '386-based PCs. A 386 can, in theory, run Linux, Emacs, and a Scheme interpreter simultaneously, but most 386s probably can't also run both Netscape and the necessary X Window System without prematurely introducing budding young underfunded hackers to the concept of *thrashing*. UTF can also fit uncompressed on a 1.44MB floppy diskette, which may come in handy for installing UTF on PCs that do not have Internet or LAN access.

The Texinfo conversion has been a straight transliteration, to the extent possible. Like the TeX-to-HTML conversion, this was not without some introduction of breakage. In the case of Unofficial Texinfo Format, figures have suffered an amateurish resurrection of the lost art of ASCII. Also, it's quite possible that some errors of ambiguity were introduced during the conversion of some of the copious superscripts ('^') and subscripts ('_'). Divining *which* has been left as an exercise to the reader. But at least we don't put our brave astronauts at risk by encoding the *greater-than-or-equal* symbol as `<u>&gt;</u>`.

If you modify `sicp.texi` to correct errors or improve the ASCII art, then update the `@set utfversion utfversion` line to reflect your delta. For example, if you started with Lytha's version `1`, and your name is Bob, then you could name your successive versions `1.bob1`, `1.bob2`, ... `1.bob`*n*. Also update `utfversiondate`. If you want to distribute your version on the Web, then embedding the string "sicp.texi" somewhere in the file or Web page will make it easier for people to find with Web search engines.

It is believed that the Unofficial Texinfo Format is in keeping with the spirit of the graciously freely-distributed HTML version. But you never know when someone's armada of lawyers might need something to do, and get their shorts all in a knot over some benign little thing, so think twice before you use your full name or distribute Info, DVI, PostScript, or PDF formats that might embed your account or machine name.

Peath, Lytha Ayth

Addendum: See also the SICP video lectures by Abelson and Sussman: at MIT CSAIL or MIT OCW.

Second Addendum: Above is the original introduction to the UTF from 2001. Ten years later, UTF has been transformed: mathematical symbols and formulas are properly typeset, and figures drawn in vector graphics. The original text formulas and ASCII art figures are still there in the Texinfo source, but will display only when compiled to Info output. At the dawn of e-book readers and tablets, reading a PDF on screen is officially not silly anymore. Enjoy!

A.R, May, 2011

Dedication

This book is dedicated, in respect and admiration, to the spirit that lives in the computer.

"I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all, I hope we don't become missionaries. Don't feel as if you're Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more."

—Alan J. Perlis (April 1, 1922 – February 7, 1990)

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

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 redesigned 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 rewritten 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 (Section 4.4), we have students use the register-machine simulator but we do not cover its implementation (Section 5.2), and we give only a cursory overview of the compiler (Section 5.5). 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 http://mitpress.mit.edu/sicp 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.

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 λ-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.

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 λ-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 Moyé 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 Carrette, 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 R^{4}RS, 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.

1

Building Abstractions with Procedures

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 Python

We need an appropriate language for describing processes, and we will use for this purpose the programming language Python. 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 procedural thoughts will be expressed in Python.

Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, *benevolent dictator for life* (BDFL).

About the origin of Python, Van Rossum wrote in 1996:

> Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office … would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Python 2.0 was released on 16 October 2000, and included many major new features including a full garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.

Python 3.0 (also called Python 3000 or py3k), a major, backwards-incompatible release, was released on 3 December 2008 after a long period of testing. Many of its major features have been backported to the backwards-compatible Python 2.6 and 2.7.

The Python Interpreter

see chapter 1 of the course notes. or Lutz

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

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

1.1.1 Expressions

One easy way to get started at programming is to examine some typical interactions with an interpreter for the Python prgramming language. Imagine that you are sitting at a computer terminal. You type an *expression*, and the interpreter responds by displaying the result of its

[1]The 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: Some computer systems distinguish *integers*, such as 2, from *real numbers*, such as 2.71. Is the real number 2.00 different from the integer 2? Are the arithmetic operations used for integers the same as the operations used for real numbers? Does 6 divided by 2 produce 3, or 3.0? How large a number can we represent? How many decimal places of accuracy can we represent? Is the range of integers the same as the range of real numbers? 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.

evaluating that expression.

One kind of primitive expression you might type is a number. (More precisely, the expression that you type consists of the numerals that represent the number in base 10.) If you present Python with a number

```
>>> 486
```

the interpreter will respond by printing[2]

```
486
```

Expressions representing numbers may be combined with an expression representing a primitive procedure (such as + or *) to form a compound expression that represents the application of the procedure to those numbers. For example:

```
>>> 137 + 349
486

>>> 1000 - 334
666

>>> 5 * 99
495

>>> 10 / 5
2

>>> 2.7 + 10
12.7
```

[2]Throughout this book, when we wish to emphasize the distinction between the input typed by the user and the response printed by the interpreter, we will show the latter in slanted characters.

Expressions such as these, formed by infixing two primitive expressions with a primitive procedure in order to denote procedure application, are called *binary combinations*. The infixed element is called the *operator*, and the other elements are called *operands*. The value of a combination is obtained by applying the procedure specified by the operator to the *arguments* that are the values of the operands.

We can also have *nested combinations*, that is, to have combinations in which the operands are themselves combinations:

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

Note that in the above expression the parentheses are optional, if you leave them out the interpreter applies the usual mathematical rules to evaluate the expression. There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions that the Lisp interpreter can evaluate. It is we humans who get confused by still relatively simple expressions such as

```
(3 * ((2 * 4) + (3 + 5))) + ((10 - 7) + 6)
```

which the interpreter would readily evaluate to be 57.

Even with complex expressions, the interpreter always operates in the same basic cycle: It reads an expression from the terminal, evaluates the expression, 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 expression.

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. We say that the name identifies a variable whose value is the object. In Python, we name things using the *assignment statement* or the `def` keyword. Typing

```
>>> size = 2
```

causes the interpreter to associate the value 2 with the name `size` . Once the name `size` has been associated with the number 2, we can refer to the value 2 by name:

```
>>> size
2

>>> 5 * size
10
```

Here are further examples of the use of assignment :

```
>>> pi = 3.14159
>>> radius = 10
>>> pi * (radius * radius)
314.159
>>> circumference = 2 * pi * radius
>>> circumference
62.8318
```

The assignment statement is our language's simplest means of abstraction, for it allows 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. The interpreter makes this step-by-step program construction particularly convenient because name-object associations can be created incrementally in successive interactions. This feature encourages the incremental development and testing of programs and is largely responsible for the fact that a Python program usually consists of a large number of relatively simple procedures.

It should be clear that the possibility of associating values with symbols 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 Combinations

One of our goals in this chapter is to isolate issues about thinking procedurally. As a case in point, let us consider that, in evaluating combinations, the interpreter is itself following a procedure.

To evaluate a combination, do the following:

1. Evaluate the subexpressions of the combination.
2. Apply the procedure (the operator) to the arguments that are the values of the other subexpressions (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 a combination we must first perform the evaluation process on each element of the 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.

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

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 + 5)
```

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 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 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*.

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

- the values of numerals are the numbers that they name,
- the values of built-in operators are the machine instruction sequences that carry out the corresponding operations, and
- the values of other names are the objects associated with those names in the environment.

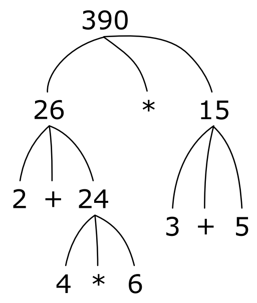


Figure 1.1: Tree representation, showing the value of each subcombination.

We may regard the second rule as a special case of the third one by stipulating that symbols such as `+` and `*` are also included in the global environment, and are associated with the sequences of machine instructions that are their "values." The key point to notice is the role of the environment in determining the meaning of the symbols in expressions. In an interactive language such as Python, 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 symbol `x` (or even for the symbol `+`). 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 assignments. For instance, evaluating `x = 3` does not apply `=` to two operands, one of which is the value of the symbol `x` and the other of which is 3, since the purpose of the assignment operator `=` is precisely to associate `x` with a value. (That is, `x = 3` is not a combination.)

Such exceptions to the general evaluation rule are called *special forms*. Assignment Statement is the only example of a special form that we have seen so far, but we will meet others shortly. Each special form has its own evaluation rule. The various kinds of expressions (each with its associated evaluation rule) constitute the syntax of the programming language.

1.1.4 Compound Procedures

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

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

Now we will learn about *procedure definitions*, 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, multiply it by itself." This is expressed in our language as

```
def square (x):
    return x * x
```

We can understand this in the following way:

```
def   square    (x):    return     (x       *         x)
 |      |         |                 |       |         |
 To   square  something,          take it  multiply  it by itself.
```

We have here a *compound procedure*, which has been given the name `square`. The procedure 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 definition creates this compound procedure and associates it with the name `square`.[4]

The general form of a procedure definition is

```
def ⟨name⟩(⟨formal parameters⟩):
  ⟨body⟩
```

The ⟨*name*⟩ is a symbol to be associated with the procedure definition in the environment.[5] The ⟨*formal parameters*⟩ are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The ⟨*body*⟩ is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied.[6] The ⟨*name*⟩ and

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

[5]Throughout this book, we will describe the general syntax of expressions by using italic symbols delimited by angle brackets—e.g., ⟨*name*⟩—to denote the "slots" in the expression to be filled in when such an expression is actually used.

[6]More generally, the body of the procedure can be a sequence of expressions. In this case, the interpreter evaluates each expression in the sequence in turn and returns the value of the expression in the return statement (if present) as the value of the procedure application.

the ⟨*formal parameters*⟩ are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

Having defined `square`, we can now use it:

```
>>> square(21)
441
>>> square (2 + 5)
49
>>> square(square (3))
81
```

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

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

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

```
def sum_of_squares(x, y):
    return square (x) + square (y)

>>> sum_of_squares(3, 4)
25
```

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

```
def f(a):
    return sum_of_squares((a + 1), (a * 2))

>>> f(5)
136
```

Compound procedures are used in exactly the same way as primitive procedures. Indeed, one could not tell by looking at the definition of `sum_of_squares` given above whether `square` was built into the interpreter, or defined as a compound procedure.

1.1.5 The Substitution Model for Procedure Application

We can assume that the mechanism for applying built-in procedures to arguments is built into the interpreter. For user-defined compound procedures, the application process is as follows:

> To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

To illustrate this process, let's evaluate the combination

```
f(5)
```

where `f` is the procedure defined in Section 1.1.4. We begin by retrieving the body of `f`:

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

Then we replace the formal parameter `a` by the argument 5:

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

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

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

If we use the definition of `square`, this reduces to

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

which reduces by multiplication to

```
36 + 100
```

and finally to

```
136
```

The process we have just described is called the *substitution model* for procedure application. It can be taken as a model that determines the "meaning" of procedure application, insofar as the procedures 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 procedure application, not to provide a description of how the interpreter really works. Typical interpreters do not evaluate procedure applications by manipulating the text of a procedure to substitute values for the formal parameters. In practice, the "substitution" is accomplished by using a local environment for the formal parameters. We will discuss this more fully in Chapter 3 and Chapter 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 procedures with "mutable data," we will see that the substitution model breaks down and must be replaced by a more complicated model of procedure application.[7]

Applicative order versus normal order

According to the description of evaluation given in Section 1.1.3, the interpreter first evaluates the operator and operands and then applies the resulting procedure 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 operand expressions for parameters until it obtained an expression involving only primitive 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)) + ((5 * 2) * (5 * 2))
```

[7]Despite 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 formal parameters of a procedure and the (possibly identical) names used in the expressions to which the procedure 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.

followed by the reductions

```
(6 * 6) +  (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 procedure applications that can be modeled using substitution (including all the procedures 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.)

Python uses applicative-order evaluation, partly because of the additional efficiency obtained from avoiding multiple evaluations of expressions such as those illustrated with `(5 + 1)` and `(5 * 2)` 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 Chapter 3 and Chapter 4.[8]

[8]In 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 evalu-

1.1.6 Conditional Expressions and Predicates

The expressive power of the class of procedures that we can define 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 define a procedure that computes the absolute value of a number by testing whether the number is positive, negative, or zero and taking different actions in the different cases according to the rule

$$|x| = \begin{cases} x & \text{if} \quad x > 0, \\ 0 & \text{if} \quad x = 0, \\ -x & \text{if} \quad x < 0. \end{cases}$$

This construct is called a *case analysis*, and there is a special form in Python for notating such a case analysis. It is called the `if` statment , and it is used as follows:

```
def abs(x):
    if (x > 0):
        return x
    elif (x == 0):
        return 0
    else:
        return -x
```

The general form of a conditional expression is

```
if (⟨p1⟩):
    (⟨e1⟩)
elif (⟨p2⟩):
    (⟨e2⟩)
      ...
```

ation. In Section 4.2 we will modify the Scheme interpreter to produce a normal-order variant of Scheme.

```
else:
    ⟨e_n⟩))
```

The `elif`(you can have zero or more of these) and `else` parts are optional. The first expression after the keyword `if` and `elif` is a *predicate*—that is, an expression whose value is interpreted as either true or false.[9]

Conditional expressions are evaluated as follows. The predicate $\langle p_1 \rangle$ is evaluated first. If its value is false, then $\langle p_2 \rangle$ is evaluated. If $\langle p_2 \rangle$'s value is also false, then $\langle p_3 \rangle$ is evaluated. This process continues until a predicate is found whose value is true, in which case the interpreter evaluates the corresponding *consequent statement* $\langle e \rangle$. If none of the $\langle p \rangle$'s is found to be true, the statement associated with the `else` part is evaluated.

The word *predicate* is used for expressions that return true or false. The absolute-value procedure `abs` makes use of the primitive comparison operators >, <, and ==.[10] These take two numbers(and other object, as we will later see) as operands and test whether the first number is, respectively, greater than, less than, or equal to the second number, returning true or false accordingly.

Another way to write the absolute-value procedure is

```
def abs(x):
    if (x < 0):
        return -x
```

[9]"Interpreted as either true or false" means this: In Python, there are two distinguished values that are denoted by the constants `Trure` and `False`. When the interpreter checks a predicate's value, it interprets some values (for example the empty string "", the empty list [], `0`, and `None`) as false. Any other value is treated as true. In this book we will use names `true` and `false`, which are associated with the values `True` and `False` respectively.

[10]`Abs` also uses the "minus" operator `-`, which, when used with a single operand, as in `-x`, indicates negation.

```
    else:
        return x
```

which could be expressed in English as “If x is less than zero return $-x$; otherwise return x.” `else` is a keyword. This causes the `if` statement to evaluate the corresponding $\langle e\rangle$ whenever all previous predicates have been evaluated to false. In fact, any expression that always evaluates to a true value could be used as the $\langle p\rangle$ of an `elif` here.

Here is yet another way to write the absolute-value procedure:

```
def abs(x):
    return -x if (x < 0) else x
```

This uses the special form `if` expression, a restricted type of conditional that can be used when there are precisely two cases in the case analysis. The general form of an `if` expression is

```
⟨consequent⟩ if ⟨predicate⟩ else ⟨alternative⟩
```

To evaluate an `if` expression, the interpreter starts by evaluating the ⟨*predicate*⟩ part of the expression. If the ⟨*predicate*⟩ evaluates to a true value, the interpreter then evaluates the ⟨*consequent*⟩ and returns its value. Otherwise it evaluates the ⟨*alternative*⟩ and returns its value.[11]

In addition to primitive predicates such as <, =, and >, there are logical composition operations, which enable us to construct compound predicates. The three most frequently used are these:

- $\langle e_1\rangle$`and` $\langle e_2\rangle$

  The interpreter evaluates the expressions $\langle e\rangle$ one at a time, in left-to-right order. If $\langle e_1\rangle$ evaluates to false, the value of the `and` ex-

[11]Note that an *if statement* does not have a value, that is you can't use it in compound expressions, an *if expression* on the other have can be placed anywhere and expression can

pression is false, and $\langle e_2 \rangle$ is not evaluated. If both $\langle e \rangle$'s evaluate to true values, the value of the `and` expression is tru.

- $\langle e_1 \rangle$`or` $\langle e_2 \rangle$

  The interpreter evaluates the expressions $\langle e \rangle$ one at a time, in left-to-right order. If any $\langle e_1 \rangle$ evaluates to a true value, that value is returned as the value of the `or` expression, and $\langle e_2 \rangle$ is not evaluated. If both $\langle e \rangle$'s evaluate to false, the value of the `or` expression is false.

- `not` $\langle e \rangle$

  The value of a `not` expression is true when the expression $\langle e \rangle$ evaluates to false, and false otherwise.

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

```
(x > 5) and (x < 10))
```

For other primitive comparison operators see Python Documentation

Exercise 1.1: Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
10
5 + 3 + 4
9 - 1
6 / 2
6 \% 4
127 // 10
(2 * 4) + (4 - 6)
```

```
a = 3
b = a + 1
a + b + a * b
a = b
if (b > a) and (b < (a * b)):
    return b
else:
    return a

if (a == 4)
    return 6
elif (b == 4):
    return (6 + 7 + a)
else:
    return 25

2 + b if (b > a) a
```

Exercise 1.2: Translate the following expression into Python:

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

Exercise 1.3: Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

Exercise 1.4: Observe that our model of evaluation allows for function applications in which the functions are compound expressions. Use this observation to describe the behavior of the following procedure:

```
def add(a, b):
    return a + b
def sub(a, b):
    return a - b

def a_plus_abs_b(a, b):
  (add if (> b 0) else sub)(a, b)
```

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 defines the following two procedures:

```
def p(a, b):
    return a/b

def test(x, y):
    if (x == 0):
        return 0
    else:
        return y
```

Then he evaluates the expression

```
(test 0 p(1,0))
```

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 `if` statement 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.)

1.1.7 Example: Square Roots by Newton's Method

Procedures, 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 procedures. Procedures must be effective.

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

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

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 procedure. Indeed, it tells us almost nothing about how to actually find the square root of a given number.

The contrast between function and procedure 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]

[12]Declarative 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

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           (2/1) = 2                ((2 + 1)/2) = 1.5
1.5         (2/1.5) = 1.3333         ((1.3333 + 1.5)/2) = 1.4167
1.4167      (2/1.4167) = 1.4118      ((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 procedures. 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 procedure:

```python
def sqrt_iter(guess, x):
    if is_good_enough(guess, x):
        return guess
```

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.

[13]This 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 Python procedure in Section 1.3.4.

```
    else:
        return sqrt_iter(improve(guess, x), x)
```

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

```
def imporve(guess, x):
    return average(guess, (x/guess))
```

where

```
def 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.) The idea is to improve the answer until it is close enough so that its square differs from the radicand by less than a predetermined tolerance (here 0.001):[14]

```
def is_good_enough(guess, x)
  return abs(square(guess) - x)) < 0.001
```

Finally, we need a way to get started. For instance, we can always guess that the square root of any number is 1:[15]

[14]We will usually give predicates names starting with `'is'`, to help us remember that they are predicates. This is just a stylistic convention.

[15]Observe that we express our initial guess as 1.0 rather than 1. This would not make any difference in some Python implementations. Python 2.7, however, distinguishes between exact integers and decimal values, and dividing two integers produces a integer number rather than a decimal. If we start with an initial guess of 1 in our square-root program, and *x* is an exact integer, all subsequent values produced in the square-root computation will be rational numbers rather than decimals. Mixed operations on rational numbers and decimals always yield decimals, so starting with an initial guess of 1.0 forces all subsequent values to be decimals.

```
def sqrt(x):
  return sqrt_iter(1.0, x)
```

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

```
sqrt(9)
3.00009155413138

sqrt(100 + 37))
11.704699917758145

sqrt(sqrt(2) + sqrt(3))
1.7739279023207892

square(sqrt(1000))
1000.000369924366
```

The `sqrt` program also illustrates that the simple procedural 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 included in our language any iterative (looping) constructs that direct the computer to do something over and over again. `Sqrt_iter`, on the other hand, demonstrates how iteration can be accomplished using no special construct other than the ordinary ability to call a procedure.[16]

> **Exercise 1.6:** Alyssa P. Hacker doesn't see why the `if` expression form needs to be provided by Python. "Why can't I just define it as an ordinary procedure in terms of ordi-

[16]Readers who are worried about the efficiency issues involved in using procedure calls to implement iteration should note the remarks on "tail recursion" in Section 1.2.1.

nary `if`?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if`:

```
def new_if(predicate, then_clause, else_clause):
   if predicate:
       then_clause
   else:
       else_clause
```

Eva demonstrates the program for Alyssa:

```
>>> new_if((2 == 3), 0, 5)
5
>>> new_if((1 == 1), 0, 5)
0
```

Delighted, Alyssa uses `new_if` to write the square-root program:

```
def sqrt_iter(guess, x):
    return new_if(is_good_enough(guess, x),
                  guess,
                  sqrt_iter(improve(guess, x), x))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

Exercise 1.7: The `is_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 `is_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 procedure that uses this kind of end test. Does this work better for small and large numbers?

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 procedure analogous to the square-root procedure. (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 procedures.)

1.1.8 Procedures as Black-Box Abstractions

`Sqrt` is our first example of a process defined by a set of mutually defined procedures. Notice that the definition of `sqrt_iter` is *recursive*; that is, the procedure is defined in terms of itself. The idea of being able to define a procedure 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

```
              sqrt
               |
           sqrt-iter
           /       \
  good-enough    improve
     /     \          \
square    abs       average
```

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

accomplished by a separate procedure. The entire `sqrt` program can be viewed as a cluster of procedures (shown in Figure 1.2) that mirrors the decomposition of the problem into subproblems.

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 procedure accomplishes an identifiable task that can be used as a module in defining other procedures. For example, when we define the `is_good_enough` procedure in terms of `square`, we are able to regard the `square` procedure as a "black box." We are not at that moment concerned with *how* the procedure 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 `is_good_enough` procedure is concerned, `square` is not quite a procedure but rather an abstraction of a procedure, a so-called *procedural abstraction.* At this level of abstraction, any procedure that computes the square is equally good.

Thus, considering only the values they return, the following two procedures for squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.[17]

```
import math
def square(x): return x * x
def double(x): return x + x
def square(x): return math.exp(double(math.log(x)))
```

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

Local names

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

```
def square(x): return x * x
def square(y): return y * y
```

This principle—that the meaning of a procedure 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 procedure must be local to the body of the procedure. For example, we used `square` in the definition of `is_good_enough` in our square-root procedure:

[17]It is not even clear which of these procedures 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.

```
def is_good_enough(guess, x)
  return abs(square(guess) - x)) < 0.001
```

The intention of the author of is_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 is_good_enough 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 is_good_enough must be a different x than the one in square. Running the procedure square must not affect the value of x that is used by is_good_enough, because that value of x may be needed by is_good_enough after square is done computing.

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

A formal parameter of a procedure has a very special role in the procedure definition, in that it doesn't matter what name the formal parameter has. Such a name is called a *bound variable*, and we say that the procedure definition *binds* its formal parameters. The meaning of a procedure definition is unchanged if a bound variable is consistently renamed throughout the definition.[18] If a variable is not bound, we say that it is *free*. The set of expressions for which a binding defines a name is called the *scope* of that name. In a procedure definition, the bound variables declared as the formal parameters of the procedure have the body of the procedure as their scope.

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

In the definition of is_good_enough above, guess and x are bound variables but <, -, abs, and square are free. The meaning of is_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 variable abs. It would have changed from free to bound.) The meaning of is_good_enough is not independent of the names of its free variables, however. It surely depends upon the fact (external to this definition) that the symbol abs names a procedure for computing the absolute value of a number. is_good_enough will compute a different function if we substitute cos for abs in its definition.

Internal definitions and block structure

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

```
def sqrt(x):
  return sqrt_iter(1.0, x)
def sqrt_iter(guess, x):
    if is_good_enough(guess, x):
        return guess
    else:
        return sqrt_iter(improve(guess, x), x)
def is_good_enough(guess, x)
  return abs(square(guess) - x)) < 0.001
def imporve(guess, x):
    return average(guess, (x/guess))
```

The problem with this program is that the only procedure that is important to users of `sqrt` is `sqrt`. The other procedures (`sqrt_iter`, `is_good_enough`, and `improve`) only clutter up their minds. They may not define any other procedure called `is_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 procedures, many numerical functions are computed as successive approximations and thus might have procedures named `is_good_enough` and `improve` as auxiliary procedures. We would like to localize the subprocedures, hiding them inside `sqrt` so that `sqrt` could coexist with other successive approximations, each having its own private `is_good_enough` procedure. To make this possible, we allow a procedure to have internal definitions that are local to that procedure. For example, in the square-root problem we can write

```
def sqrt(x):
    def is_good_enough(guess, x)
        return abs(square(guess) - x)) < 0.001
    def imporve(guess, x):
        return average(guess, (x/guess))
    def sqrt_iter(guess, x):
        if is_good_enough(guess, x):
            return guess
        else:
            return sqrt_iter(improve(guess, x), x)
    return sqrt_iter(1.0, x)
```

Such nesting of definitions, called *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 definitions of the auxiliary procedures, we can simplify them. Since `x` is bound in the definition of sqrt, the procedures is_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 procedures. Instead, we allow x to be a free variable in the internal definitions, as shown below. Then x gets its value from the argument with which the enclosing procedure sqrt is called. This discipline is called *lexical scoping*.[19]

```
def sqrt(x):
    def is_good_enough(guess)
        return abs(square(guess) - x)) < 0.001
    def imporve(guess):
        return average(guess, (x/guess))
    def sqrt_iter(guess):
        if is_good_enough(guess):
            return guess
        else:
            return 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.[20] 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.

[19]Lexical scoping dictates that free variables in a procedure are taken to refer to bindings made by enclosing procedure definitions; that is, they are looked up in the environment in which the procedure was defined. We will see how this works in detail in chapter 3 when we study environments and the detailed behavior of the interpreter.

[20]Embedded definitions must come first in a procedure body. The management is not responsible for the consequences of running programs that intertwine definition and use.

1.2 Procedures 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 defining them as compound procedures. 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 procedures are worth defining). We lack the experience to predict the consequences of making a move (executing a procedure).

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 development 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 procedures. Only after we have developed such a skill can we learn to reliably construct programs that exhibit the desired behavior.

A procedure 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 procedure. 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 procedures. We will also investigate the rates at which these processes consume the important computational resources of time and space. The procedures 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.

1.2.1 Linear Recursion and Iteration

We begin by considering the factorial function, defined by

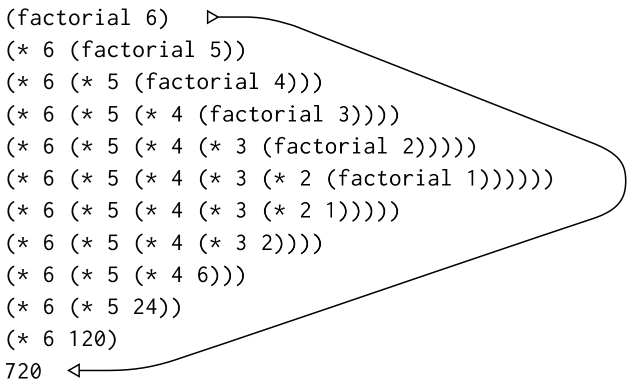
$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1.$$

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 procedure:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```



```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

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

We can use the substitution model of Section 1.1.5 to watch this procedure in action computing 6!, as shown in Figure 1.3.

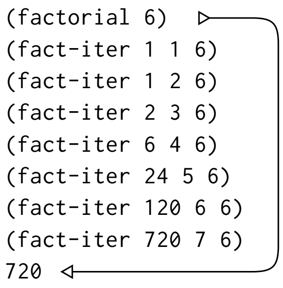
Now let's take a different perspective on computing factorials. We could describe a rule for computing *n*! by specifying that we first multiply 1 by 2, then multiply the result by 3, then by 4, and so on until we reach *n*. More formally, we maintain a running product, together with a counter that counts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

```
product ← counter * product
counter ← 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 procedure for computing factorials:[21]

[21] In a real program we would probably use the block structure introduced in the last section to hide the definition of `fact_iter`:



```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

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

```scheme
(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

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

```scheme
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (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 variables `product`, `counter`, and `max_count`. We call this an *iterative process*. In general, an iterative process is one whose state can be summarized by a fixed number of *state variables*, together with a fixed rule that describes how the state variables 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 program variables 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 program variables. Not so with the recursive process. In this case there is some additional "hidden" information, maintained by the interpreter and not contained in the program variables, which indicates "where the process is" in negotiating the chain of deferred operations. The longer the chain, the more information must be maintained.[22]

In contrasting iteration and recursion, we must be careful not to confuse the notion of a recursive *process* with the notion of a recursive *procedure*. When we describe a procedure as recursive, we are referring to the syntactic fact that the procedure definition refers (either directly or indirectly) to the procedure itself. But when we describe a process as following a pattern that is, say, linearly recursive, we are speaking about how the process evolves, not about the syntax of how a procedure is written. It may seem disturbing that we refer to a recursive procedure such as `fact_iter` as generating an iterative process. However, the process really is iterative: Its state is captured completely by its three state variables, and an interpreter need keep track of only three variables 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 procedure consumes an amount of memory that

[22]When we discuss the implementation of procedures 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*.

grows with the number of procedure 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 Scheme 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 procedure. An implementation with this property is called *tail-recursive*. With a tail-recursive implementation, iteration can be expressed using the ordinary procedure call mechanism, so that special iteration constructs are useful only as syntactic sugar.[23]

Exercise 1.9: Each of the following two procedures defines a method for adding two positive integers in terms of the procedures `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
(define (+ a b)
  (if (= a 0) b (inc (+ (dec a) b))))
(define (+ a b)
  (if (= a 0) b (+ (dec a) (inc b))))
```

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

[23]Tail 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 and Sussman 1975) constructed a tail-recursive interpreter for Scheme. Steele later showed how tail recursion is a consequence of the natural way to compile procedure calls (Steele 1977). The IEEE standard for Scheme requires that Scheme implementations be tail-recursive.

Exercise 1.10: The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (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 procedures, where `A` is the procedure defined above:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

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

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

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

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

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Consider the pattern of this computation. To compute `(fib 5)`, we compute `(fib 4)` and `(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` procedure calls itself twice each time it is invoked.

This procedure 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 procedure will compute `(fib 1)` or `(fib 0)` (the number of leaves in the above tree, in general) is precisely $\mathrm{Fib}(n+1)$. To get an idea of how bad this is, one can show that the value of $\mathrm{Fib}(n)$ grows exponentially with n. More precisely (see Exercise 1.13), $\mathrm{Fib}(n)$ is the closest integer to $\varphi^n/\sqrt{5}$, where

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180$$

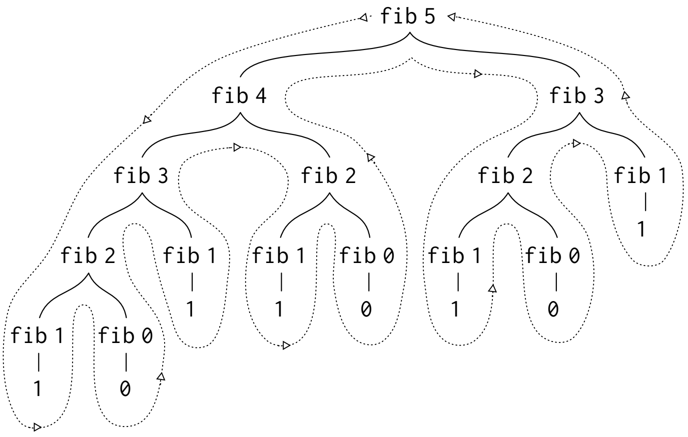


Figure 1.5: The tree-recursive process generated in computing (`fib 5`).

is the *golden ratio*, which satisfies the equation

$$\varphi^2 = \varphi + 1.$$

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 Fib(n + 1) and Fib(n). Thus, we can compute Fibonacci numbers iteratively using the procedure

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

This second method for computing 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 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.[24] 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` procedure is much less efficient than the second one, it is more straightforward, being little more than a translation into Lisp 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 variables.

[24]An example of this was hinted at in Section 1.1.3. The interpreter itself evaluates expressions using a tree-recursive process.

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 procedure to compute the number of ways to change any given amount of money?

This problem has a simple solution as a recursive procedure. 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 the problem of changing smaller amounts using fewer kinds of coins. Consider this reduction rule carefully, and convince yourself that we can use it to describe an algorithm if we specify the following degenerate cases:[25]

- 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 procedure:

```
(define (count-change amount) (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination
                         kinds-of-coins))
                     kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
  (cond ((= 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)))
```

(The `first_denomination` procedure 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:

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

```
(count-change 100)
292
```

`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, 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 procedures into more efficient procedures that compute the same result.[26]

> **Exercise 1.11:** A function f is defined by the rule that
>
> $$f(n) = \begin{cases} n \quad \text{if} \quad n < 3, \\ f(n-1) + 2f(n-2) + 3f(n-3) \quad \text{if} \quad n \geq 3. \end{cases}$$
>
> Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

> **Exercise 1.12:** The following pattern of numbers is called *Pascal's triangle.*

[26]One 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 procedure 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.

```
        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.[27] Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

Exercise 1.13: Prove that Fib(n) is the closest integer to $\varphi^n/\sqrt{5}$, where $\varphi = (1+\sqrt{5})/2$. Hint: Let $\psi = (1-\sqrt{5})/2$. Use induction and the definition of the Fibonacci numbers (see Section 1.2.2) to prove that $\text{Fib}(n) = (\varphi^n - \psi^n)/\sqrt{5}$.

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.

[27]The 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 Bháscara Áchárya.

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 $k_1 f(n) \leq R(n) \leq k_2 f(n)$ for any sufficiently large value of n. (In other words, for large n, the value $R(n)$ is sandwiched between $k_1 f(n)$ and $k_2 f(n)$.)

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.[28] The tree-recursive Fibonacci computation requires $\Theta(\varphi^n)$

[28]These 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.

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 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` procedure 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?

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 procedures:

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
   (if (not (> (abs angle) 0.1))
       angle
       (p (sine (/ angle 3.0)))))
```

a. How many times is the procedure `p` applied when `(sine 12.15)` is evaluated?

b. 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` procedure when `(sine a)` is evaluated?

1.2.4 Exponentiation

Consider the problem of computing the exponential of a given number. We would like a procedure 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 procedure

```
(define (expt b n)
  (if (= 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:

```
(define (expt b n)
  (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= 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

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

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 procedure:

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

where the predicate to test whether an integer is even is defined in terms of the primitive procedure `remainder` by

```
(define (even? n)
  (= (remainder 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.[29]

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.[30] 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.[31]

Exercise 1.16: Design a procedure that evolves an iterative exponentiation process that uses successive squaring

[29]More 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 $\Theta(\log n)$.

[30]You may wonder why anyone would care about raising numbers to the 1000th power. See Section 1.2.6.

[31]This iterative algorithm is ancient. It appears in the *Chandah-sutra* by Áchárya Pingala, 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.

and uses a logarithmic number of steps, as does `fast_expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable 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.)

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 procedure (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* 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 procedure analogous to `fast_expt` that uses a logarithmic number of steps.

Exercise 1.18: Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.[32]

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 variables 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 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` procedure. Put this all together to complete the following procedure, which runs in a logarithmic

[32]This 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.

number of steps:[33]

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   ⟨??⟩     ; compute p′
                   ⟨??⟩     ; compute q′
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

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.

[33]This 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

```
GCD(a,b) = 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,

```
GCD(206,40) = GCD(40,6)
            = GCD(6,4)
            = GCD(4,2)
            = GCD(2,0)
            = 2
```

reduces GCD(206, 40) to 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*.[34]

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

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

[34]Euclid'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.

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.[35]

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 procedure. If the process takes k steps, then we must have $n \geq \text{Fib}(k) \approx \varphi^k/\sqrt{5}$. Therefore

[35]This 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 $a_{k-1} = b_k, b_{k-1} =$ remainder of a_k 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 $k = 1$, since this merely requires that b be at least as large as 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.

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 procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` procedure given above. Suppose we were to interpret this procedure 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 `(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 `(gcd 206 40)`? In the applicative-order evaluation?

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.

```
(define (smallest-divisor n) (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (divides? a b) (= (remainder 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.

```
(define (prime? n)
  (= 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 $\sqrt{n}$.[36] 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.[37]

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

[37] Pierre 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

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

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

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m))
          m))
```

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.

```
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m))))
```

This is very similar to the `fast_expt` procedure of Section 1.2.4. It uses successive squaring, so that the number of steps grows logarithmically with the exponent.[38]

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 procedure `random`, which we assume is included as a primitive in Scheme. `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:

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

The following procedure 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.

```
(define (fast-prime? n times)
```

[38]The 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.)

```
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

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.[39]

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

[39] 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.

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.[40]

> **Exercise 1.21:** Use the `smallest_divisor` procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

> **Exercise 1.22:** Most Lisp implementations include a primitive called `runtime` that returns an integer that specifies the amount of time the system has been running (measured, for

[40]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 et al. (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.

example, in microseconds). The following `timed_prime_test` procedure, when called with an integer n, prints n and checks to see if n is prime. If n is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))
(define (report-prime elapsed-time)
  (display " *** ")
  (display elapsed-time))
```

Using this procedure, write a procedure `search_for_primes` that checks the primality of consecutive odd integers in a specified range. Use your procedure 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 $\Theta(\sqrt{n})$ 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` procedure shown at the start of this section does lots of needless testing: 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 procedure `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest_divisor` procedure 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` procedure of Exercise 1.22 to use `fast_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

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Is she correct? Would this procedure serve as well for our fast prime tester? Explain.

Exercise 1.26: Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast_prime?` test seems to run more slowly than his `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` procedure to use an explicit multiplication, rather than calling `square`:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                       (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* 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 procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

Exercise 1.27: Demonstrate that the Carmichael numbers listed in Footnote 1.47 really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

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` procedure. 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` procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to `fermat_test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return `0`.

1.3 Formulating Abstractions with Higher-Order Procedures

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

```
(define (cube x) (* 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 defining this procedure, 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. Procedures provide this ability. This is why all but the most primitive programming languages include mechanisms for defining procedures.

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

1.3.1 Procedures as Arguments

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

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

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

```
(define (sum-cubes a b)
  (if (> 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):[41]

[41]This 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.

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
         (pi-sum (+ a 4) b))))
```

These three procedures clearly share a common underlying pattern. They are for the most part identical, differing only in the name of the procedure, 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 procedures by filling in slots in the same template:

```
(define (⟨name⟩ a b)
  (if (> 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 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 procedure that expresses the concept of summation itself rather than only procedures that compute particular sums. We can do so readily in our procedural language by taking the common template shown above and transforming the "slots" into formal parameters:

```
(define (sum term a next b)
  (if (> 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 procedures `term` and `next`. We can use `sum` just as we would any procedure. For example, we can use it (along with a procedure `inc` that increments its argument by 1) to define `sum_cubes`:

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (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)
3025
```

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

```
(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b))
```

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

```
(sum-integers 1 10)
55
```

We can also define pi_sum in the same way:[42]

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

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

```
(* 8 (pi-sum 1 1000))
3.139592655589783
```

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 procedure:

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))

(integral cube 0 1 0.01)
.24998750000000042
```

[42]Notice that we have used block structure (Section 1.1.8) to embed the definitions of pi_next and pi_term within pi_sum, since these procedures are unlikely to be useful for any other purpose. We will see how to get rid of them altogether in Section 1.3.2.

```
(integral cube 0 1 0.001)
.249999875000001
```

(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
>
> $$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n),$$
>
> where $h = (b - a)/n$, for some even integer n, and $y_k = f(a + kh)$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f, a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate `cube` between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` procedure shown above.

> **Exercise 1.30:** The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if ⟨??⟩
        ⟨??⟩
        (iter ⟨??⟩ ⟨??⟩)))
  (iter ⟨??⟩ ⟨??⟩))
```

Exercise 1.31:

a. The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures.[43] Write an analogous procedure 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 formula[44]

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

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

Exercise 1.32:

a. Show that `sum` and `product` (Exercise 1.31) are both special cases of a still more general notion called `accumulate`

[43]The intent of Exercise 1.31 through Exercise 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.

[44]This formula was discovered by the seventeenth-century English mathematician John Wallis.

that combines a collection of terms, using some general accumulation function:

```
(accumulate combiner null-value term a next b)
```

`Accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` procedure (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 defined as simple calls to `accumulate`.

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

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 procedure. 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 `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 $\text{GCD}(i, n) = 1$).

1.3.2 Constructing Procedures Using Lambda

In using sum as in Section 1.3.1, it seems terribly awkward to have to define trivial procedures such as pi_term and pi_next just so we can use them as arguments to our higher-order procedure. Rather than define pi_next and pi_term, it would be more convenient to have a way to directly specify "the procedure that returns its input incremented by 4" and "the procedure that returns the reciprocal of its input times its input plus 2." We can do this by introducing the special form lambda, which creates procedures. Using lambda we can describe what we want as

```
(lambda (x) (+ x 4))
```

and

```
(lambda (x) (/ 1.0 (* x (+ x 2))))
```

Then our pi_sum procedure can be expressed without defining any auxiliary procedures as

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
```

Again using lambda, we can write the integral procedure without having to define the auxiliary procedure add_dx:

```
(define (integral f a b dx)
(* (sum f
        (+ a (/ dx 2.0))
        (lambda (x) (+ x dx))
        b)
   dx))
```

In general, `lambda` is used to create procedures in the same way as `define`, except that no name is specified for the procedure:

```
(lambda (⟨formal-parameters⟩) ⟨body⟩)
```

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

```
(define (plus4 x) (+ x 4))
```

is equivalent to

```
(define plus4 (lambda (x) (+ x 4)))
```

We can read a `lambda` expression as follows:

```
(lambda                     (x)     (+    x     4))
    |                        |       |    |      |
the procedure of an argument x that adds x and 4
```

Like any expression that has a procedure as its value, a `lambda` expression can be used as the operator in a combination such as

```
((lambda (x y z) (+ x y (square z)))
 1 2 3)
12
```

or, more generally, in any context where we would normally use a procedure name.[45]

[45]It would be clearer and less intimidating to people learning Lisp if a name more obvious than `lambda`, such as `make_procedure`, were used. But the convention is firmly

Using `let` to create local variables

Another use of `lambda` is in creating local variables. We often need local variables in our procedures other than those that have been bound as formal parameters. For example, suppose we wish to compute the function

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

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 procedure to compute f, we would like to include as local variables 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 procedure to bind the local variables:

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

entrenched. The notation is adopted from 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.

Of course, we could use a `lambda` expression to specify an anonymous procedure for binding our local variables. The body of `f` then becomes a single call to that procedure:

```
(define (f x y)
  ((lambda (a b)
     (+ (* x (square a))
        (* y b)
        (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

This construct is so useful that there is a special form called `let` to make its use more convenient. Using `let`, the `f` procedure could be written as

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

The general form of a `let` expression is

```
(let ((⟨var₁⟩ ⟨exp₁⟩)
      (⟨var₂⟩ ⟨exp₂⟩)
      ...
      (⟨varₙ⟩ ⟨expₙ⟩))
   ⟨body⟩)
```

which can be thought of as saying

```
let ⟨var₁⟩ have the value ⟨exp₁⟩ and
    ⟨var₂⟩ have the value ⟨exp₂⟩ and
    ...
    ⟨varₙ⟩ have the value ⟨expₙ⟩
in  ⟨body⟩
```

The first part of the let expression is a list of name-expression pairs. When the let is evaluated, each name is associated with the value of the corresponding expression. The body of the let is evaluated with these names bound as local variables. The way this happens is that the let expression is interpreted as an alternate syntax for

```
((lambda (⟨var₁⟩ ... ⟨varₙ⟩)
    ⟨body⟩)
 ⟨exp₁⟩
 ...
 ⟨expₙ⟩)
```

No new mechanism is required in the interpreter in order to provide local variables. A let expression is simply syntactic sugar for the underlying lambda application.

We can see from this equivalence that the scope of a variable specified by a let expression is the body of the let. This implies that:

- Let allows one to bind variables as locally as possible to where they are to be used. For example, if the value of x is 5, the value of the expression

  ```
  (+ (let ((x 3))
       (+ x (* x 10)))
     x)
  ```

  is 38. Here, the x in the body of the let is 3, so the value of the let expression is 33. On the other hand, the x that is the second argument to the outermost + is still 5.

- The variables' values are computed outside the let. This matters when the expressions that provide the values for the local variables depend upon variables having the same names as the local variables themselves. For example, if the value of `x` is 2, the expression

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

will have the value 12 because, inside the body of the `let`, `x` will be 3 and `y` will be 4 (which is the outer `x` plus 2).

Sometimes we can use internal definitions to get the same effect as with `let`. For example, we could have defined the procedure `f` above as

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x (square a))
     (* y b)
     (* a b)))
```

We prefer, however, to use `let` in situations like this and to use internal `define` only for internal procedures.[46]

Exercise 1.34: Suppose we define the procedure

```
(define (f g) (g 2))
```

Then we have

[46]Understanding internal definitions well enough to be sure a program means what we intend it to mean requires a more elaborate model of the evaluation process than we have presented in this chapter. The subtleties do not arise with internal definitions of procedures, however. We will return to this issue in Section 4.1.6, after we learn more about evaluation.

```
(f square)
4
(f (lambda (z) (* z (+ z 1))))
6
```

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

1.3.3 Procedures as General Methods

We introduced compound procedures 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 procedures, such as the `integral` procedure of Section 1.3.1, we began to see a more powerful kind of abstraction: procedures 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 procedures.

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 procedure that implements this strategy:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else 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 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 procedure similar to the one used in Section 1.1.7 for computing square roots:[47]

```
(define (close-enough? x y) (< (abs (- x y)) 0.001))
```

`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 procedure, which checks to see which of the endpoints has a negative function value and which has a positive value, and calls the `search` procedure accordingly. If the function has the same sign on the two given points, the half-interval method cannot be used, in which case the procedure signals an error.[48]

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Values are not of opposite sign" a b)))))
```

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 sin 2.0 4.0)
3.14111328125
```

[47]We 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.

[48]This can be accomplished using `error`, which takes as arguments a number of items that are printed as error messages.

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 (lambda (x) (- (* x x x) (* 2 x) 3))
                      1.0
                      2.0)
1.89306640625
```

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,

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

until the value does not change very much. Using this idea, we can devise a procedure `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:

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try 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:[49]

```
(fixed-point cos 1.0)
.7390822985224023
```

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

```
(fixed-point (lambda (y) (+ (sin y) (cos y)))
             1.0)
1.2587315962971173
```

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 function[50] $y \mapsto x/y$, and we can therefore try to compute square roots as

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

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

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

[50]$\mapsto$ (pronounced "maps to") is the mathematician's way of writing `lambda`. $y \mapsto x/y$ means `(lambda (y) (/ x y))`, that is, the function whose value at y is x/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 $y \mapsto \frac{1}{2}(y + x/y)$:

```
(define (sqrt x)
  (fixed-point (lambda (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 procedure 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 procedure of Section 1.1.7. This approach of averaging successive approximations to a solution, a technique that 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` procedure.

> **Exercise 1.36:** Modify `fixed_point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives 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 Scheme's primitive `log` procedure, 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.)

Exercise 1.37:

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

```
(cont-frac (lambda (i) 1.0)
           (lambda (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?

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

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` procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

Exercise 1.39: A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

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

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

1.3.4 Procedures as Returned Values

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

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 procedure 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 procedure:

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

`Average_damp` is a procedure that takes as its argument a procedure `f` and returns as its value a procedure (produced by the `lambda`) that, when applied to a number `x`, produces the average of `x` and `(f x)`. For example, applying `average_damp` to the `square` procedure produces a procedure whose value at some number x is the average of x and x^2. Applying this resulting procedure to 10 returns the average of 10 and

100, or 55:[51]

```
((average-damp square) 10)
55
```

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

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (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 procedures 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 procedure. Experienced programmers know how to choose procedural 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 procedure to one that extracts cube roots:[52]

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

[51]Observe 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 procedure that is obtained as the value returned by a higher-order procedure.

[52]See Exercise 1.45 for a further generalization.

Newton’s method

When we first introduced the square-root procedure, 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 function f.[53]

For many functions g and for sufficiently good initial guesses for x, Newton’s method converges very rapidly to a solution of $g(x) = 0$.[54]

In order to implement Newton’s method as a procedure, 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

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

[53]Elementary 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.

[54]Newton’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.

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

```
(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
```

along with the definition

```
(define dx 0.00001)
```

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

```
(define (cube x) (* x x x))
((deriv cube) 5)
75.00014999664018
```

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

```
(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

The `newton_transform` procedure expresses the formula at the beginning of this section, and `newtons_method` is readily defined in terms of this. It takes as arguments a procedure 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 $y \mapsto y^2 - x$ starting with an initial guess of 1.[55]

This provides yet another form of the square-root procedure:

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

```
(define (sqrt x)
  (newtons-method
   (lambda (y) (- (square y) x)) 1.0))
```

Abstractions and first-class procedures

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 procedure:

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

This very general procedure takes as its arguments a procedure g that computes some function, a procedure 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:

```
(define (sqrt x)
  (fixed-point-of-transform
   (lambda (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

```
(define (sqrt x)
  (fixed-point-of-transform
  (lambda (y) (- (square y) x)) newton-transform 1.0))
```

We began Section 1.3 with the observation that compound procedures are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming language. Now we've seen how higher-order procedures 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 procedures 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:[56]

- They may be named by variables.
- They may be passed as arguments to procedures.
- They may be returned as the results of procedures.

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

- They may be included in data structures.[57]

Lisp, unlike other common programming languages, awards procedures full first-class status. This poses challenges for efficient implementation, but the resulting gain in expressive power is enormous.[58]

> **Exercise 1.40:** Define a procedure `cubic` that can be used together with the `newtons_method` procedure in expressions of the form
>
> ```
> (newtons-method (cubic a b c) 1)
> ```
>
> to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

> **Exercise 1.41:** Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by
>
> ```
> (((double (double double)) inc) 5)
> ```

> **Exercise 1.42:** Let f and g be two one-argument functions. The *composition* f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

[57] We'll see examples of this after we introduce data structures in Chapter 2.

[58] The major implementation cost of first-class procedures is that allowing procedures to be returned as values requires reserving storage for a procedure's free variables even while the procedure is not executing. In the Scheme implementation we will study in Section 4.1, these variables are stored in the procedure's environment.

```
((compose square inc) 6)
49
```

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 procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

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

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 procedure `smooth` that takes as input a procedure that computes f and returns a procedure 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.

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 procedure for computing n^{th} roots using `fixed_point`, `average_damp`, and the `repeated` procedure of Exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

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 procedure `iterative_improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `Iterative_improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of Section 1.1.7 and the `fixed_point` procedure of Section 1.3.3 in terms of `iterative_improve`.

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 procedures in program design. We saw how to use primitive data (numbers) and primitive operations (arithmetic operations), how to combine procedures to form compound procedures through composition, conditionals, and the use of parameters, and how to abstract procedures by using `def`. We saw that a procedure 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 procedures. We also saw that higher-order procedures 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 procedures 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 procedures to form compound procedures, 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 procedures: 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 define procedures 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 procedures (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 procedure 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 define the procedure

```
def 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 express, in procedural terms, the idea that one can form linear combinations whenever addition and multiplication are defined—for rational numbers, complex numbers, polynomials, or whatever. We could express this as a procedure of the form

```
def linear-combination(a, b, x, y):
    return add(mul(a, x), mul(b, y))
```

where `add` and `mul` are not the primitive procedures `+` 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 procedures `add` and `mul` will perform the appropriate manipulations. From the perspective of the procedure `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 procedure such as `linear_combination` to pass its arguments along to `add` and `mul` without having to know their detailed structure.[1]

[1]The ability to directly manipulate procedures provides an analogous increase in the expressive power of a programming language. For example, in Section 1.3.1 we introduced the `sum` procedure, which takes a procedure `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 procedure 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 procedure," 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.

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 procedures, 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 procedures. This will further blur the distinction between "procedure" 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 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 procedure used as an element in creating a more complex procedure could be regarded not only as a collection of particular operations but also as a procedural abstraction. That is, the details of how the procedure was implemented could be suppressed, and the particular procedure itself could be replaced by any other procedure with the same overall behavior. In other words, we could make an abstraction that would separate the way the procedure would be used from the details of how the procedure would be implemented in terms of more primitive procedures. 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 procedures, 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 procedures 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 procedures:

- `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 procedures `numer`, `denom`, and `make_rat` should be implemented. Even so, if we did have these three procedures, we could then add, subtract, multiply, divide, and test equality by using the following relations:

$$\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}, \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}, \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2}, \\
\frac{n_1/d_1}{n_2/d_2} &= \frac{n_1 d_2}{d_1 n_2}, \\
\frac{n_1}{d_1} &= \frac{n_2}{d_2} \quad \text{if and only if} \quad n_1 d_2 = n_2 d_1.
\end{aligned}$$

We can express these rules as procedures:

```
def add_rat(x, y):
    return make_rat( (numer(x)*denom(y))+(numer(y)*denom(x)),
                     (denom(x)*denom(y)))


def sub_rat(x, y):
    return make_rat( (numer(x)*denom(y))-(numer(y)*denom(x)),
                     (denom(x)*denom(y)))


def mul_rat(x, y):
    return make_rat( numer(x)*numer(y),
                     denom(x)*denom(y))


def div_rat(x, y):
    return make_rat( numer(x)*denom(y),
                     denom(x)*numer(y))
```

```
def isEqual_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 procedures `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 primitive construct of forming a `tuple`. This construct takes two arguments and returns a compound data object that contains the two arguments as parts. For example to form a pair of numbers 1 and 2 named `p` execute the following code:

```
>>> p = (1, 2)
>>> type(p)
<class 'tuple'>
```

Given a pair, we can extract the parts using the `[ ]` notation:

```
>>> p[0]
1
>>> p[1]
2
```

For our purpose we will form a conventional functional abstraction that uses the underlying facility provided by Python:

```
def cons(a, b):
    return (a, b)
```

```
def car(x):
    return x[0]

def cdr(x):
    return x[1]
```

So now we have a procedure `cons` that 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 procedures `car` and `cdr`.[2] Thus, we can use `cons`, `car`, and `cdr` as follows:

```
>>> x = cons(1, 2)
>>> car(x)
1
>>> cdr(x)
2
```

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

```
>>> x = cons(1, 2)
>>> y = cons(3, 4)
>>> z = cons(x, y)
>>> car(car(z))
1
>>> car(cdr(z))
3
```

[2]The name `cons` stands for "construct." The names `car` and `cdr` derive from the original implementation of Lisp on the IBM 704. That machine had an addressing scheme that allowed one to reference the "address" and "decrement" parts of a memory location. `Car` stands for "Contents of Address part of Register" and `cdr` (pronounced "could-er") stands for "Contents of Decrement part of Register."

In Section 2.2 we will see how this ability to combine pairs means that pairs can be used as general-purpose building blocks to create all sorts of complex data structures. The single compound-data primitive *pair*, implemented by the procedures `cons`, `car`, and `cdr`, 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]

```
def make_rat(n, d):
    return cons(n, d)

def numer(r):
    return car(r)

def denom(r):
    return cdr(r)
```

[3]Another way to define the selectors and constructor is

```
make_rat = cons
numer = car
denom = cdr
```

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

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

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

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]

```
def print_rat(r):
    print("%d/%d" % (numer(r),denom(r)))
```

Now we can try our rational-number procedures:

```
>>> one_half = make_rat(1, 2)
>>> print_rat(one_half)
1/2
>>> one_third = make_rat(1, 3)
>>> print_rat(add_rat(one_half, one_third))
5/6
>>> print_rat(mul_rat(one_half, one_third))
1/6
>>> print_rat(add_rat(one_third, one_third))
6/9
```

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` procedure 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:

```
def make_rat(n, d):
    div = gcd(n, d)
    return (n/div, d/div)
```

[4]`print` is the Scheme primitive for printing data. The Scheme primitive
`n` starts a new line for printing. Neither of these procedures returns a useful value, so in the uses of `print_rat` below, we show only what `print_rat` prints, not what the interpreter prints as the value returned by `print_rat`.

Now we have

```
>>> print_rat(add_rat(one_third, one_third))
2/3
```

as desired. This modification was accomplished by changing the constructor `make_rat` without changing any of the procedures (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. `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.

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 procedures

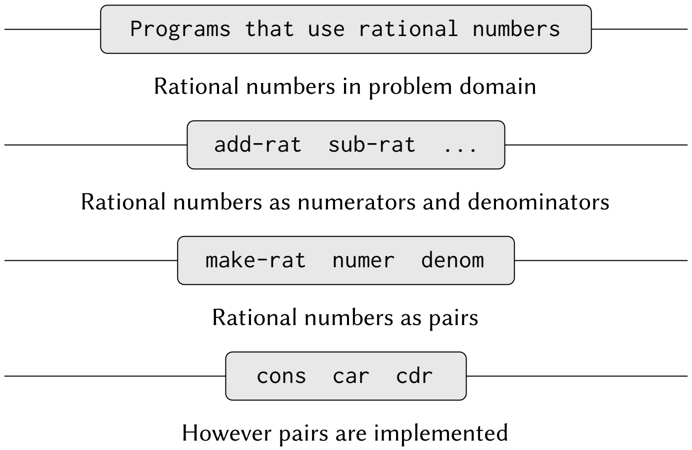

Programs that use rational numbers
Rational numbers in problem domain
add-rat sub-rat ...
Rational numbers as numerators and denominators
make-rat numer denom
Rational numbers as pairs
cons car cdr
However pairs are implemented

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

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 `cons`, `car`, and `cdr`. In effect, procedures at each level are the interfaces that define the abstraction barriers and connect the different levels.

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 procedures:

```
def make_rat(n, d):
    return (n, d)

def numer(x):
    g = gcd(car(x), cdr(x))
    return car(x)/g

def denom(x):
    g = gcd(car(x), cdr(x))
    return cdr(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 procedures `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 procedures 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. Define 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, define a procedure `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 procedures, you'll need a way to print points:

```
def print_point(p):
    print('(',x_point(p), y_point(p),')')
```

> **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 procedures 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 procedures will work using either representation?

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 procedures: `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 procedures.

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 procedures 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

$$\frac{\texttt{(numer x)}}{\texttt{(denom x)}} = \frac{\texttt{n}}{\texttt{d}}\,.$$

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 procedures must fulfill in order to be a valid representation.[5]

[5]Surprisingly, this idea is very difficult to formulate rigorously. There are two ap-

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 have to say what a pair is, only that the language supplies procedures `cons`, `car`, and `cdr` 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 `cons` we can retrieve the objects using `car` and `cdr`. That is, the operations satisfy the condition that, for any objects `x` and `y`, if `z` is `cons(x  y)` then `car( z)` is `x` and `cdr(z)` is `y`. Indeed, we can mention that these three procedures are included as primitives in our language. However, any triple of procedures that satisfies the above condition can be used as the basis for implementing pairs. This point is illustrated strikingly by the fact that we could implement `cons`, `car`, and `cdr` without using any data structures at all but only using procedures. Here are the definitions:

```
def cons(x, y):
    def dispatch(m):
        if (m == 0):
```

proaches to giving such a formulation. One, pioneered by C. A. R. Hoare (1972), is known as the method of *abstract models*. It formalizes the "procedures 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 et al. 1978), and by Guttag at Toronto (see Guttag 1977), is called *algebraic specification*. It regards the "procedures" 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).

```
            return x
        elif (m == 1):
            return y
        else:
            print('error: Argument not 0 or 1')
    return dispatch

def car(r):
    return r(0)

def cdr(r):
    return r(1)
```

This use of procedures 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 procedures satisfy the condition given above.

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

The point of exhibiting the procedural representation of pairs is not that our language works this way (Python and other language systems in general, implement pairs directly, for efficiency reasons) but that it could work this way. The procedural 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 procedures as objects automatically provides the ability to represent compound data. This may seem a curiosity now, but procedural 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 procedural representation of pairs. For this representation, verify that `car(cons( x y))` yields `x` for any objects `x` and `y`.

```
def cons(x, y):
    return lambda f: f(x, y)

def car(z):
    return z(lambda p,q: p)
```

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

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 procedures `cons`, `car`, and `cdr`.

Exercise 2.6: In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
zero = lambda f: lambda x: x
def add1(n):
    return lambda f: lambda 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 procedure `+` (not in terms of repeated application of `add_1`).

2.1.4 Extended Exercise: Interval Arithmetic (SKIP THIS FOR NOW)

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, 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 procedure 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:

```
(define (add-interval x y)
  (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. (`Min` and `max` are primitives that find the minimum or maximum of any number of arguments.)

```
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
    (make-interval (min p1 p2 p3 p4)
                   (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.

```
(define (div-interval x y)
  (mul-interval
   x
   (make-interval (/ 1.0 (upper-bound y))
                  (/ 1.0 (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:

```
(define (make-interval a b) (cons a b))
```

Define selectors `upper_bound` and `lower_bound` to complete the implementation.

Exercise 2.8: Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction procedure, called `sub_interval`.

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.

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 code to check for this condition and to signal an error if it occurs.

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 procedure using Ben's suggestion.

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:

```
(define (make-center-width c w)
  (make-interval (- c w) (+ c w)))
(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))
(define (width i)
  (/ (- (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.

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.

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:

```
(define (par1 r1 r2)
  (div-interval (mul-interval r1 r2)
                (add-interval r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (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 Exercise 2.12).

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 variable that represents an uncertain number is repeated. Thus, she says, `par2` is a "better" program for parallel resistances than `par1`. Is she right? Why?

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.)

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 `(cons 1 2)`. In this representation, which is called *box-and-pointer notation*, each object is shown as a *pointer* to a box. The box for a primitive object contains a representation of the object. For example, the box for a number contains a numeral. The box for a pair is actually a double box, the left part containing (a pointer to) the `car` of the pair and the right part containing the `cdr`.

We have already seen that `cons` can be used to combine not only numbers but pairs as well. (You made use of this fact, or should have, in doing Exercise 2.2 and Exercise 2.3.) As a consequence, pairs provide a universal building block from which we can construct all sorts of

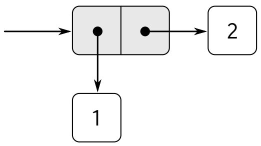
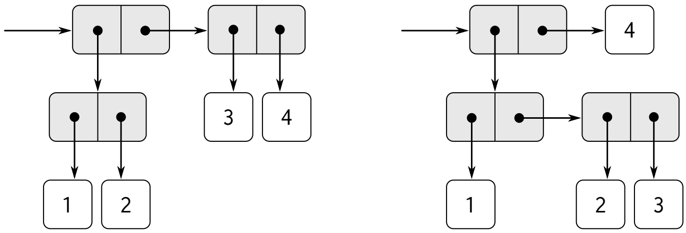


Figure 2.2: Box-and-pointer representation of `(cons 1 2)`.



```
(cons (cons 1 2)
      (cons 3 4))
```

```
(cons (cons 1
            (cons 2 3))
      4)
```

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

data structures. Figure 2.3 shows two ways to use pairs to combine the numbers 1, 2, 3, and 4.

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 `cons`. 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.[6] Closure is the key to power in any means of combina-

[6]The use of the word "closure" here comes from abstract algebra, where a set of

tion 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 procedures, 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.[7]

2.2.1 Representing Sequences

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

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 procedures with free variables. We do not use the word "closure" in this second sense in this book.

[7]The 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."

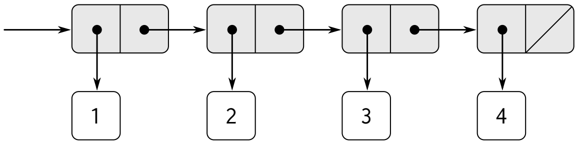


Figure 2.4: The sequence 1, 2, 3, 4 represented as a chain of pairs.

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 `car` of each pair is the corresponding item in the chain, and the `cdr` of the pair is the next pair in the chain. The `cdr` 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 the variable `nil`. The entire sequence is constructed by nested `cons` operations:

```
cons(1,
      cons(2,
             cons(3,
                   cons(4, None))))
```

Such a sequence of pairs, formed by nested conses, is called a *list*. Note that Python provides a primitive list data structure for constructing lists. But here we are constructing our own list data structure. The following function lets us easily construct list using `cons`

```
def make_list(*args):
    if args == ():
        return None
    else:
        return cons(args[0], make_list(*(args[1:])))
```

We will show the lists by printing the sequence of elements, enclosed in parentheses. Thus, the data object in Figure 2.4 is printed as `(1 2 3 4)`[8]:

```
one_through_four = make_list(1 2 3 4)
print_list(one_through_four)
(1 2 3 4)
```

Note: For the rest of the chapter we are not going to show the application of `print_list` to a list for showing the output.

We can think of `car` as selecting the first item in the list, and of `cdr` as selecting the sublist consisting of all but the first item. Nested applications of `car` and `cdr` can be used to extract the second, third, and subsequent items in the list.

The value of `None`, used to terminate the chain of pairs, can be thought of as a sequence of no elements, the *empty list*.[9]

[8] A sample application of printing out the list can be as follows:

```
def print_list(lst):
    def helper(lst, pos):
        if not isPair(lst):
            if lst == None:
                print(")", end=' ')
            else:
                print(lst, end=' ')
        else:
            if pos=='car':
                print("(", end=' ')
            helper(car(lst), 'car')
            helper(cdr(lst), 'cdr')
    helper(lst, 'car')
```

[9] It's remarkable how much energy in the standardization of Lisp dialects has been dissipated in arguments that are literally over nothing: Should `nil` be an ordinary

List operations

The use of pairs to represent sequences of elements as lists is accompanied by conventional programming techniques for manipulating lists by successively "cdring down" the lists. For example, the procedure 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 car of the list.
- Otherwise, list_ref should return the $(n - 1)$-st item of the cdr of the list.

```
def list_ref(items, n):
    if n == 0:
        return car(items)
    else:
        return list_ref(cdr(items), n-1)

>>> squares = make_list(1, 4, 9, 16, 25)
>>> list_ref(squares, 3)
16
```

Often we cdr down the whole list. To aid in this let us define a predicate isNull, which tests whether its argurment is the empty list.

name? Should the value of nil be a symbol? Should it be a list? Should it be a pair? In Scheme, nil is an ordinary name, which we use in this section as a variable whose value is the end-of-list marker (just as true is an ordinary variable that has a true value). Other dialects of Lisp, including Common Lisp, treat nil as a special symbol. The authors of this book, who have endured too many language standardization brawls, would like to avoid the entire issue. Once we have introduced quotation in Section 2.3, we will denote the empty list as '() and dispense with the variable nil entirely.

```
def isNull(lst):
    return lst == None
```

The procedure `length`, which returns the number of items in a list, illustrates this typical patter of use:

```
def length(items):
    if isNull(items):
        return 0
    else:
        return 1 + length(cdr(items))

>>> odds = make_list(1, 3, 5, 7, 9)
>>> length(squares)
5
```

The `length` procedure implements a simple recursive plan. The reduction step is:

- The `length` of any list is 1 plus the `length` of the `cdr` 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:

```
def length(items):
    def length_iter(a, count):
        if isNull(a):
            return count
        else:
            return length_iter(cdr(a), 1+count)
    return length_iter(items, 0)
```

Another conventional programming technique is to "cons up" an answer list while cdring down a list, as in the procedure append, which takes two lists as arguments and combines their elements to make a new list:

```
append(squares odds)
```

should give you a list represented graphically by:

```
(1 4 9 16 25 1 3 5 7)
```

Similarly

```
(append odds squares)
```

should give you the list

```
(1 3 5 7 1 4 9 16 25)
```

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 cdr of `list1` and `list2`, and cons the car of `list1` onto the result:

```
def append(list1, list2):
    if isNull(list1):
        return list2
    else:
        return cons(car(list1), append(cdr(list1), list2))
```

Exercise 2.17: Define a procedure `last_pair` that returns the list that contains only the last element of a given (nonempty) list:

Exercise 2.18: Define a procedure `reverse` that takes a list as argument and returns a list of the same elements in reverse order:

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 procedure `first_denomination` and partly into the procedure `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 procedure `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:

```
us_coins = make_list(50, 25, 10, 5, 1)
uk_coins = make_list(100, 50, 20, 10, 5, 2, 1, 0.5)
```

We could then call `cc` as follows:

```
cc(100, us-coins)
292
```

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:

```
def cc(amount, coin_values):
        if amount == 0:
            return 1
        elif (amount<0) or no_more(coin_values):
            return 0
        else:
            return cc(amount, excpt_fst_denom(coin_values)) + \
                   cc(amount - fst_denom(coin_values), coin_values)
```

Define the procedures `fst_denom`, which returns the first denomination in the coin_values, `excpt_fst_denom`, which returns all the coins except the first, 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?

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 procedure scales each number in a list by a given factor:

```
def scale_list(items, factor):
    if isNull(items):
        None
    else:
        cons( (car(items)* factor), scale_list(cdr(items), factor))

>>> scale_list(make_list(1,2,3,4,5), 10)
(10 20 30 40 50)
```

We can abstract this general idea and capture it as a common pattern expressed as a higher-order procedure, just as in Section 1.3. The higher-order procedure here is called `map`. `Map` takes as arguments a procedure of one argument and a list, and returns a list of the results produced by applying the procedure to each element in the list:[10]

```
def map(f, items)
    if isNull(items):
        None
    else:
        cons(f(car(items)), map(f, cdr(items)))

>>> map(abs, make_list(-10, 2.5, -11.6, 17))
(10 2.5 11.6 17)
>>> map(lambda x: x**x, make_list(1, 2, 3, 4))
(1 4 9 16)
```

Now we can give a new definition of `scale_list` in terms of `map`:

```
def scale-list(items, factor):
    return map(lambda x: x * factor, items)
```

`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

[10] Python 2.7 provides built-in function called `map` that works like the `map` function defined above. In Python 3.x `map` now returns an iterator. For example:

```
>>> list(map(lambda x: x**3, range(1, 6)))
[1, 8, 27, 64, 125]
>>> for i in map(lambda x: x**2, range(1, 4)):
        print(i, end=' ')

1 4 9
```

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 procedures 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 procedure `square_list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
square_list(make_list(1, 2, 3, 4))
(1 4 9 16)
```

Here are two different definitions of `square_list`. Complete both of them by filling in the missing expressions:

```
def square_list(items):
    if isNull(items):
        None
    else:
        cons( ⟨??⟩, ⟨??⟩)

def square_list(items)
    map( ⟨??⟩, ⟨??⟩)
```

Exercise 2.22: Louis Reasoner tries to rewrite the first `square_list` procedure of Exercise 2.21 so that it evolves an iterative process:

```
def square_list(items):
    def iter(things, answer):
        if isNull(things):
            return answer
        else:
            return iter(cdr(things), cons(square(car(things)), answer)
    return iter(items, None)
```

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 cons:

```
def square_list(items):
    def iter(things, answer):
        if isNull(things):
            return answer
        else:
            return iter(cdr(things), cons(answer, square(car(things)))
    return iter(items, None)
```

This doesn't work either. Explain.

Exercise 2.23: The procedure for_each is similar to map. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, for_each just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all—for_each is used with procedures that perform an action, such as printing. For example,

```
for_each(lambda x: print(x), make_list(57, 321, 88))
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`.

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) 3, 4)` constructed by

```
cons( make_list(1, 2), make_list(3, 4))
```

as a list of three items, the first of which is itself a list, `(1 2)`. Indeed, this is suggested by the form in which the result is printed by the interpreter. Figure 2.5 shows the representation of this structure in terms of pairs.

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.

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` procedure of Section 2.2.1 with the `count_leaves` procedure, which returns the total number of leaves of a tree:

```
t = cons( make_list(1, 2), make_list(3, 4) )
length(t)
```

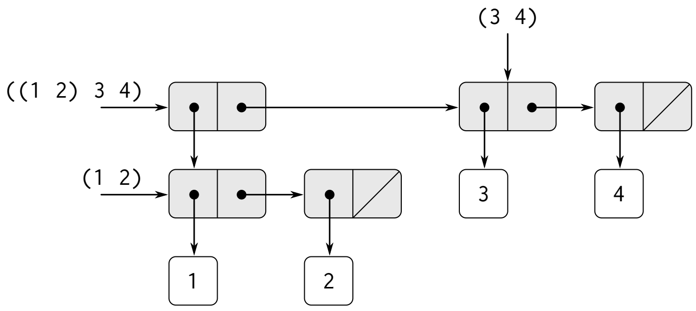


Figure 2.5: Structure formed by `cons( make_list(1, 2), make_list(3, 4))`.

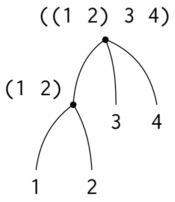


Figure 2.6: The list structure in Figure 2.5 viewed as a tree.

```
3
count_leaves(t)
4
tt = make_list(t, t)
(((1 2) 3 4) ((1 2) 3 4))
length(tt)
2
count_leaves(tt)
8
```

To implement `count_leaves`, recall the recursive plan for computing `length`:

- `Length` of a list `x` is 1 plus `length` of the `cdr` of `x`.
- `Length` of the empty list is 0.

`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 `car` of the list, we must take into account that the `car` 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 `car` of `x` plus `count_leaves` of the `cdr` of `x`.

Finally, by taking `car`s we reach actual leaves, so we need another base case:

- `Count_leaves` of a leaf is 1.

To aid in writing recursive procedures on trees, let us provide a predicate `isPair`, which tests whether its argument is a pair(made using `cons`):

```
def isPair(obj):
    return isinstance(obj, tuple)

>>> isPair(cons(1, None))
True
>>> isPair(4)
False
```

Here is the complete procedure:[11]

```
def count_leaves(x):
    if isNull(x):
        return 0
    elif not isPair(x):
        return 1
    else:
        return count_leaves(car(x)) + \
               count_leaves(cdr(x))
```

Exercise 2.24: Suppose we evaluate the expression `make_list(1, make_list(2, make_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).

Exercise 2.25: Give combinations of `car`s and `cdr`s that will pick 7 from each of the following lists:

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

Exercise 2.26: Suppose we define `x` and `y` to be two lists:

```
x = make_list(1, 2, 3)
y = make_list(4, 5, 6)
```

What result is printed by the interpreter in response to evaluating each of the following expressions:

[11]The order of the first two clauses in the cond matters, since the empty list satisfies `isNull` and also is not a pair.

```
append(x, y)
cons(x, y)
make_list(x, y)
```

Exercise 2.27: Modify your `reverse` procedure of Exercise 2.18 to produce a `deep_reverse` procedure 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,

```
x = make_list( make_list(1, 2), make_list(3, 4) )
x
((1 2) (3 4))
(reverse x)
((3 4) (1 2))
(deep_reverse x)
((4 3) (2 1))
```

Exercise 2.28: Write a procedure `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,

```
x = make_list( make_list(1, 2), make_list(3, 4) )
fringe(x)
(1 2 3 4)
fringe(make_list(x, x))
(1 2 3 4 1 2 3 4)
```

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`):

```
(define (make-mobile left right)
  (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:

```
(define (make-branch length structure)
  (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 procedure `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

```
(define (make-mobile left right) (cons left right))
(define (make-branch length structure)
  (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

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 procedure, 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:

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))
(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)) 10)
(10 (20 (30 40) 50) (60 70))
```

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:

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
         (if (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 procedure square_tree analogous to the square_list procedure 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)))
(1 (4 (9 16) 25) (36 49))
```

Define square_tree both directly (i.e., without using any higher-order procedures) and also by using map and recursion.

Exercise 2.31: Abstract your answer to Exercise 2.30 to produce a procedure tree_map with the property that square_tree could be defined as

```
(define (square-tree tree) (tree-map square tree))
```

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), then the set of all subsets is (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3)). Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map ⟨??⟩ rest)))))
```

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 procedures, 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 procedure, analogous to the `count_leaves` procedure of Section 2.2.2, which takes a tree as argument and computes the sum of the squares of the leaves that are odd:

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

On the surface, this procedure is very different from the following one, which constructs a list of all the even Fibonacci numbers Fib(k), where *k* is less than or equal to a given integer *n*:

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

Despite the fact that these two procedures 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 `cons`, starting with the empty list.

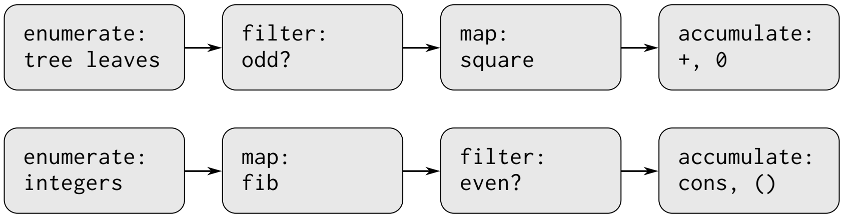


Figure 2.7: The signal-flow plans for the procedures `sum_odd_squares` (top) and `even_fibs` (bottom) reveal the commonality between the two programs.

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` procedure 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.

Unfortunately, the two procedure definitions above fail to exhibit this signal-flow structure. For instance, if we examine the `sum_odd_squares` procedure, we find that the enumeration is implemented partly by the `null?` and `pair?` tests and partly by the tree-recursive structure of the procedure. 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 procedure that correspond to the elements in the signal-flow description. Our two procedures 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 procedures 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` procedure from Section 2.2.1:

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)
```

Filtering a sequence to select only those elements that satisfy a given predicate is accomplished by

```
(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

For example,

```
(filter odd? (list 1 2 3 4 5))
(1 3 5)
```

Accumulations can be implemented by

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
(accumulate + 0 (list 1 2 3 4 5))
15
(accumulate * 1 (list 1 2 3 4 5))
120
(accumulate cons nil (list 1 2 3 4 5))
(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:

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low (enumerate-interval (+ low 1) high))))
(enumerate-interval 2 7)
(2 3 4 5 6 7)
```

To enumerate the leaves of a tree, we can use[12]

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))
```

[12]This is, in fact, precisely the `fringe` procedure from Exercise 2.28. Here we've renamed it to emphasize that it is part of a family of general sequence-manipulation procedures.

```
(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

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:

```
(define (sum-odd-squares tree)
  (accumulate
   + 0 (map square (filter 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:

```
(define (even-fibs n)
  (accumulate
   cons
   nil
   (filter 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.

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 procedures in a program that constructs a list of the squares of the first $n + 1$ Fibonacci numbers:

```
(define (list-fib-squares n)
  (accumulate
   cons
   nil
   (map square (map fib (enumerate-interval 0 n)))))
(list-fib-squares 10)
(0 1 1 4 9 25 64 169 441 1156 3025)
```

We can rearrange the pieces and use them in computing the product of the squares of the odd integers in a sequence:

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate * 1 (map square (filter odd? sequence))))
(product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```

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 programmer? that tests if a record is for a programmer. Then we can write

```
(define (salary-of-highest-paid-programmer records)
  (accumulate max 0 (map salary (filter programmer? records))))
```

These examples give just a hint of the vast range of operations that can be expressed as sequence operations.[13]

[13]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

Sequences, implemented here as lists, serve as a conventional interface that permits us to combine 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:

```
(define (map p sequence)
  (accumulate (lambda (x y) ⟨??⟩) nil sequence))
(define (append seq1 seq2)
  (accumulate cons ⟨??⟩ ⟨??⟩))
(define (length sequence)
  (accumulate ⟨??⟩ 0 sequence))
```

Exercise 2.34: Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

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.

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\dots(a_n x + a_{n-1})x + \dots + a_1)x + a_0.$$

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.[14]

Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (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))
```

[14]According 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 $a_n x^n$, 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 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.

Exercise 2.35: Redefine count_leaves from Section 2.2.2 as an accumulation:

```
(define (count-leaves t)
  (accumulate ⟨??⟩ ⟨??⟩ (map ⟨??⟩ ⟨??⟩)))
```

Exercise 2.36: The procedure accumulate_n is similar to accumu_late 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 procedure 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, ((1 2 3) (4 5 6) (7 8 9) (10 11 12)), then the value of (accumulate_n + 0 s) should be the sequence (22 26 30). Fill in the missing expressions in the following definition of accumulate_n:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init ⟨??⟩)
            (accumulate-n op init ⟨??⟩))))
```

Exercise 2.37: Suppose we represent vectors $\mathbf{v} = (v_i)$ as sequences of numbers, and matrices $\mathbf{m} = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. 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 $\Sigma_i v_i w_i$,
`(matrix-*-vector m v)`	returns the vector **t**, where $t_i = \Sigma_j m_{ij} v_j$,
`(matrix-*-matrix m n)`	returns the matrix **p**, where $p_{ij} = \Sigma_k m_{ik} n_{kj}$,
`(transpose m)`	returns the matrix **n**, where $n_{ij} = m_{ji}$.

We can define the dot product as[15]

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate_n` is defined in Exercise 2.36.)

```
(define (matrix-*-vector m v)
  (map ⟨??⟩ m))
(define (transpose mat)
  (accumulate-n ⟨??⟩ ⟨??⟩ mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map ⟨??⟩ m)))
```

[15]This definition uses the extended version of `map` described in Footnote 12.

Exercise 2.38: The accumulate procedure 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:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

What are the values of

```
(fold-right / 1 (list 1 2 3))
(fold-left / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left list nil (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.

Exercise 2.39: Complete the following definitions of reverse (Exercise 2.18) in terms of fold_right and fold_left from Exercise 2.38:

```
(define (reverse sequence)
  (fold-right (lambda (x y) ⟨??⟩) nil sequence))
(define (reverse sequence)
  (fold-left (lambda (x y) ⟨??⟩) nil sequence))
```

Nested Mappings

We can extend the sequence paradigm to include many computations that are commonly expressed using nested loops.[16] 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
$i + j$	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:[17]

[16]This 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.

[17]We're representing a pair here as a list of two elements rather than as a Lisp pair. Thus, the "pair" (i, j) is represented as `(list i j)`, not `(cons i j)`.

```
(accumulate
 append nil (map (lambda (i)
                   (map (lambda (j) (list i j))
                        (enumerate-interval 1 (- i 1))))
                 (enumerate-interval 1 n)))
```

The combination of mapping and accumulating with append is so common in this sort of program that we will isolate it as a separate procedure:

```
(define (flatmap proc seq)
  (accumulate append nil (map proc 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

```
(define (prime-sum? pair)
  (prime? (+ (car pair) (cadr pair))))
```

Finally, generate the sequence of results by mapping over the filtered pairs using the following procedure, which constructs a triple consisting of the two elements of the pair along with their sum:

```
(define (make-pair-sum pair)
  (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
```

Combining all these steps yields the complete procedure:

```
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum? (flatmap
                           (lambda (i)
                             (map (lambda (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 $S - x$,[18] 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:[19]

```
(define (permutations s)
  (if (null? s)                    ; empty set?
      (list nil)                   ; sequence containing empty set
      (flatmap (lambda (x)
                 (map (lambda (p) (cons 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 nil)`, which is a sequence with one item, namely the set with no elements. The `remove` procedure used in `permutations` returns all the items in a given sequence except for a given item. This can be expressed as a simple filter:

[18]The set $S - x$ is the set of all elements of S, excluding x.

[19]Semicolons in Scheme code are used to introduce *comments*. Everything from the semicolon 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.

```
(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
          sequence))
```

Exercise 2.40: Define a procedure 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.

Exercise 2.41: Write a procedure 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.

Exercise 2.42: 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,

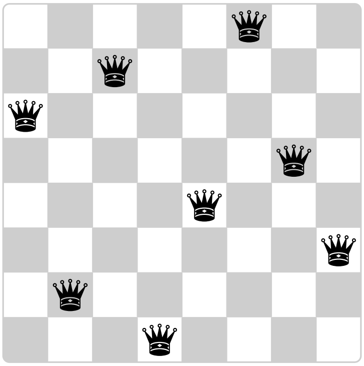


Figure 2.8: A solution to the eight-queens puzzle.

we will produce not only one solution, but all solutions to the puzzle.

We implement this solution as a procedure queens, which returns a sequence of all solutions to the problem of placing n queens on an $n \times n$ chessboard. Queens has an internal procedure queen_cols that returns the sequence of all ways to place queens in the first k columns of the board.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                    (adjoin-position
                     new-row k rest-of-queens))
                  (enumerate-interval 1 board-size)))
           (queen-cols (- k 1))))))
   (queen-cols board-size))
```

In this procedure 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 procedure 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 procedure 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.)

Exercise 2.43: Louis Reasoner is having a terrible time doing Exercise 2.42. His queens procedure 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
 (lambda (new-row)
   (map (lambda (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.

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 procedures 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.[20] In this language, the data objects being combined are represented as procedures rather than as list structure. Just as `cons`, 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.

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 lan-

[20]The 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` procedure in this section.

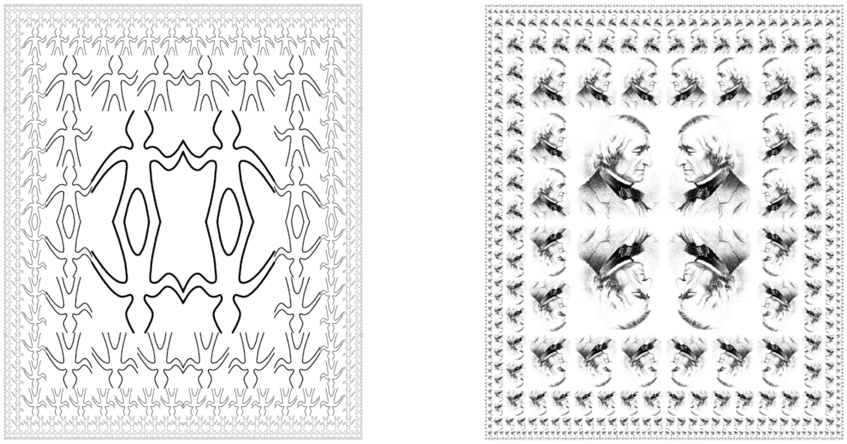


Figure 2.9: Designs generated with the picture language.

guage'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 `wave` that makes a crude line drawing, as shown in Figure 2.10. The actual shape of the drawing depends on the frame—all four images in figure 2.10 are produced by the same `wave` painter, but with respect to four different frames. Painters can be more elaborate than this: The primitive painter called `rogers` paints a picture of MIT's founder, William Barton Rogers, as shown in Figure 2.11.[21] The four images in figure 2.11 are drawn with respect to

[21]William Barton Rogers (1804-1882) was the founder and first president of MIT. A geologist and talented teacher, he taught at William and Mary College and at the

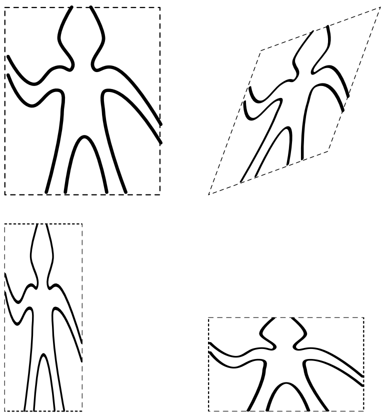


Figure 2.10: Images produced by the wave painter, with respect to four different frames. The frames, shown with dotted lines, are not part of the images.

University of Virginia. In 1859 he moved to Boston, where he had more time for research, worked on a plan for establishing a "polytechnic institute," and served as Massachusetts's first State Inspector of Gas Meters.

When MIT was established in 1861, Rogers was elected its first president. Rogers espoused an ideal of "useful learning" that was different from the university education of the time, with its overemphasis on the classics, which, as he wrote, "stand in the way of the broader, higher and more practical instruction and discipline of the natural and social sciences." This education was likewise to be different from narrow trade-school education. In Rogers's words:

> The world-enforced distinction between the practical and the scientific worker is utterly futile, and the whole experience of modern times has demonstrated its utter worthlessness.

Rogers served as president of MIT until 1870, when he resigned due to ill health. In 1878 the second president of MIT, John Runkle, resigned under the pressure of a the same four frames as the `wave` images in figure 2.10.

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, `below` 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`

---

financial crisis brought on by the Panic of 1873 and strain of fighting off attempts by Harvard to take over MIT. Rogers returned to hold the office of president until 1881.

Rogers collapsed and died while addressing MIT's graduating class at the commencement exercises of 1882. Runkle quoted Rogers's last words in a memorial address delivered that same year:

> "As I stand here today and see what the Institute is, . . . I call to mind the beginnings of science. I remember one hundred and fifty years ago Stephen Hales published a pamphlet on the subject of illuminating gas, in which he stated that his researches had demonstrated that 128 grains of bituminous coal – " "Bituminous coal," these were his last words on earth. Here he bent forward, as if consulting some notes on the table before him, then slowly regaining an erect position, threw up his hands, and was translated from the scene of his earthly labors and triumphs to "the tomorrow of death," where the mysteries of life are solved, and the disembodied spirit finds unending satisfaction in contemplating the new and still unfathomable mysteries of the infinite future.

In the words of Francis A. Walker (MIT's third president):

> All his life he had borne himself most faithfully and heroically, and he died as so good a knight would surely have wished, in harness, at his post, and in the very part and act of public duty.

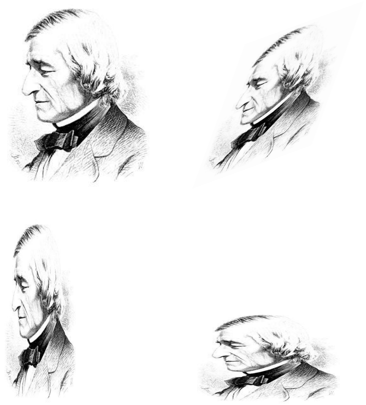


Figure 2.11: Images of William Barton Rogers, founder and first president of MIT, painted with respect to the same four frames as in Figure 2.10 (original image from Wikimedia Commons).

produces a painter that draws the original painter's image left-to-right reversed.

Figure 2.12 shows the drawing of a painter called `wave4` that is built up in two stages starting from `wave`:

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```

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 `below` 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 `cons`, the closure of our data under the means of

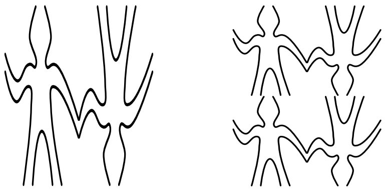


Figure 2.12: Creating a complex figure, starting from the `wave` painter of Figure 2.10.

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 Scheme procedures. This means that we don't need a special abstraction mechanism in the picture language: Since the means of combination are ordinary Scheme procedures, we automatically have the capability to do anything with painter operations that we can do with procedures. For example, we can abstract the pattern in `wave4` as

```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))
```

and define `wave4` as an instance of this pattern:

```
(define wave4 (flipped-pairs wave))
```

We can also define recursive operations. Here's one that makes painters split and branch towards the right as shown in Figure 2.13 and Figure 2.14:

```
(define (right-split painter n)
  (if (= n 0)
```

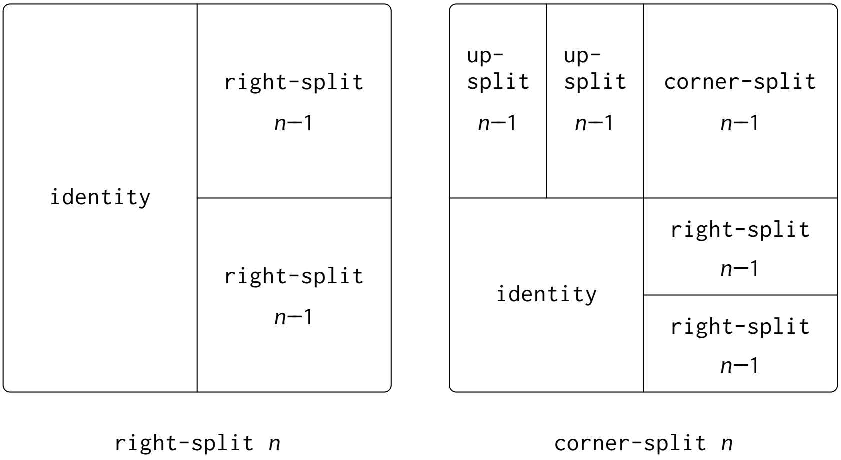


Figure 2.13: Recursive plans for `right_split` and `corner_split`.

```
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```

We can produce balanced patterns by branching upwards as well as towards the right (see exercise Exercise 2.44 and figures Figure 2.13 and Figure 2.14):

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))))
          (beside (below painter top-left)
                    (below bottom-right corner))))))
```

By placing four copies of a `corner_split` appropriately, we obtain a pattern called `square_limit`, whose application to `wave` and `rogers` is shown in Figure 2.9:

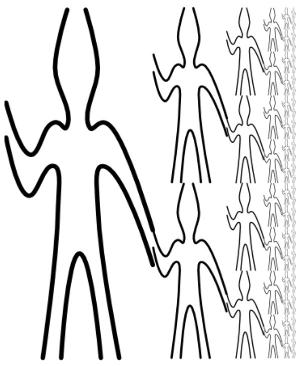
```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```

Exercise 2.44: Define the procedure `up_split` used by `corner_split`. It is similar to `right_split`, except that it switches the roles of `below` and `beside`.

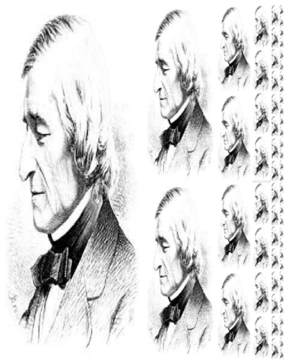
Higher-order operations

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—procedures that take painter operations as arguments and create new painter operations.

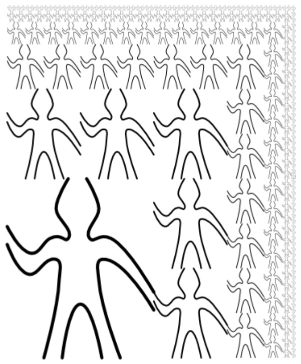
For example, `flipped_pairs` and `square_limit` each arrange four copies of a painter's image in a square pattern; they differ only in how they orient the copies. One way to abstract this pattern of painter combination is with the following procedure, which takes four one-argument painter operations and produces a painter operation that transforms a given painter with those four operations and arranges the results in a square. `Tl`, `tr`, `bl`, and `br` are the transformations to apply to the top



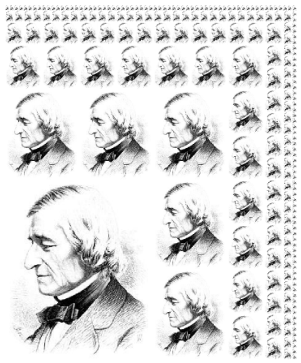
`(right-split wave 4)`



`(right-split rogers 4)`



`(corner-split wave 4)`



`(corner-split rogers 4)`

Figure 2.14: The recursive operations `right_split` and `corner_split` applied to the painters `wave` and `rogers`. Combining four `corner_split` figures produces symmetric `square_limit` designs as shown in Figure 2.9.

left copy, the top right copy, the bottom left copy, and the bottom right copy, respectively.

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

Then `flipped_pairs` can be defined in terms of `square_of_four` as follows:[22]

```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                  identity flip-vert)))
    (combine4 painter)))
```

and `square_limit` can be expressed as[23]

```
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-horiz identity
                                  rotate180 flip-vert)))
    (combine4 (corner-split painter n))))
```

Exercise 2.45: `Right_split` and `up_split` can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

[22]Equivalently, we could write

```
(define flipped-pairs
  (square-of-four identity flip-vert identity flip-vert))
```

[23]`Rotate180` rotates a painter by 180 degrees (see Exercise 2.50). Instead of `rotate180` we could say `(compose flip_vert flip_horiz)`, using the `compose` procedure from Exercise 1.42.

```
(define right-split (split beside below))
(define up-split (split below beside))
```

produces procedures `right_split` and `up_split` with the same behaviors as the ones already defined.

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).

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

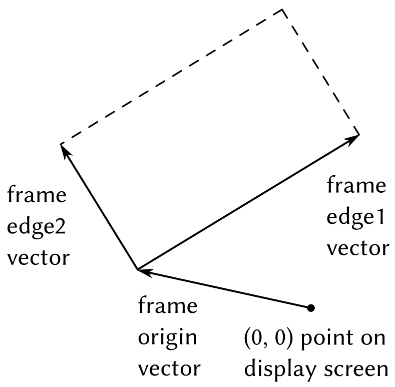


Figure 2.15: A frame is described by three vectors — an origin and two edges.

frame. We can create a frame's coordinate map with the following procedure:[24]

```
(define (frame-coord-map frame)
  (lambda (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 procedure 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,

[24]`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.

```
((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 procedures `add_vect`, `sub_vect`, and `scale_vect` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

$$\begin{aligned}(x_1, y_1) + (x_2, y_2) &= (x_1 + x_2, y_1 + y_2),\\ (x_1, y_1) - (x_2, y_2) &= (x_1 - x_2, y_1 - y_2),\\ s \cdot (x, y) &= (sx, sy).\end{aligned}$$

Exercise 2.47: Here are two possible constructors for frames:

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

For each constructor supply the appropriate selectors to produce an implementation for frames.

Painters

A painter is represented as a procedure 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 procedure 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:[25]

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
     (lambda (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 procedures 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 procedure can serve as a painter, provided that it takes a frame as argument and draws some-

[25]Segments_>painter uses the representation for line segments described in Exercise 2.48 below. It also uses the for_each procedure described in Exercise 2.23.

thing scaled to fit the frame.[26]

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`.

Exercise 2.49: Use `segments_>painter` to define the following primitive painters:

a. The painter that draws the outline of the designated frame.
b. The painter that draws an "X" by connecting opposite corners of the frame.
c. The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.
d. The `wave` painter.

[26]For example, the `rogers` painter of Figure 2.11 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 procedures. Painters are our second example of a procedural representation for data.

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 procedure `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, the first 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.

```
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (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:

```
(define (flip-vert painter)
  (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:

```
(define (shrink-to-upper-right painter)
  (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 degrees[27]

```
(define (rotate90 painter)
  (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:[28]

```
(define (squash-inwards painter)
  (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` procedure, 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.

[27]`Rotate90` is a pure rotation only for square frames, because it also stretches and shrinks the image to fit into the rotated frame.

[28]The diamond-shaped images in Figure 2.10 and Figure 2.11 were created with `squash_inwards` applied to `wave` and `rogers`.

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:

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter
            painter1
            (make-vect 0.0 0.0)
            split-point
            (make-vect 0.0 1.0)))
          (paint-right
           (transform-painter
            painter2
            split-point
            (make-vect 1.0 0.0)
            (make-vect 0.5 1.0))))
      (lambda (frame)
        (paint-left frame)
        (paint-right frame)))))
```

Observe how the painter data abstraction, and in particular the representation of painters as procedures, makes `beside` easy to implement. The `beside` procedure 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.

> **Exercise 2.51:** Define the `below` operation for painters. `Below` 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 `below` in two different ways—first by writing a procedure that is analogous to the `beside` procedure given above, and again in terms of `beside` and suitable rotation operations (from Exercise 2.50).

Levels of language for robust design

The picture language exercises some of the critical ideas we've introduced about abstraction with procedures and data. The fundamental data abstractions, painters, are implemented using procedural 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 procedures 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.[29] 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 are created using a language that specifies points and lines to provide the lists of line segments for `segments_>painter`, or the shading details for a painter like `rogers`. The bulk of our description of the picture language focused on combining these primitives, using geometric combiners such as `beside` and `below`. We also worked at a higher level, regarding `beside` and `below` 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 `wave` shown in Figure 2.9. We could work at the lowest level to change the detailed appearance of the `wave` 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.

[29]Section 3.3.4 describes one such language.

Exercise 2.52: Make changes to the square limit of `wave` shown in Figure 2.9 by working at each of the levels described above. In particular:

a. Add some segments to the primitive `wave` painter of Exercise 2.49 (to add a smile, for example).
b. 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).
c. Modify the version of `square_limit` that uses `square_of_four` so as to assemble the corners in a different pattern. (For example, you might make the big Mr. Rogers look outward from each corner of the square.)

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 arbitrary symbols as data.

2.3.1 Quotation

If we can form compound data using symbols, we can have lists such as

```
(a b c d)
(23 45 17)
((Norah 12) (Molly 9) (Anna 7) (Lauren 6) (Charlotte 4))
```

Lists containing symbols can look just like the expressions of our language:

```
(* (+ 23 45)
   (+ x 9))
(define (fact n)
  (if (= n 1) 1 (* n (fact (- n 1)))))
```

In order to manipulate symbols we need a new element in our language: the ability to *quote* a data object. Suppose we want to construct the list `(a b)`. We can't accomplish this with `(list a b)`, because this expression constructs a list of the *values* of `a` and `b` rather than the symbols themselves. This issue is well known in the context of natural languages, where words and sentences may be regarded either as semantic entities or as character strings (syntactic entities). The common practice in natural languages is to use quotation marks to 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.[30]

We can follow this same practice to identify lists and symbols that are to be treated as data objects rather than as expressions to be evalu-

[30]Allowing 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."

ated. However, our format for quoting differs from that of natural languages in that we place a quotation mark (traditionally, the single quote symbol `'`) only at the beginning of the object to be quoted. We can get away with this in Scheme syntax because we rely on blanks and parentheses to delimit objects. Thus, the meaning of the single quote character is to quote the next object.[31]

Now we can distinguish between symbols and their values:

```
(define a 1)
(define b 2)
(list a b)
(1 2)
(list 'a 'b)
(a b)
(list 'a b)
(a 2)
```

Quotation also allows us to type in compound objects, using the conventional printed representation for lists:[32]

---

[31]The single quote is different from the double quote we have been using to enclose character strings to be printed. Whereas the single quote can be used to denote lists or symbols, the double quote is used only with character strings. In this book, the only use for character strings is as items to be printed.

[32]Strictly, our use of the quotation mark violates the general rule that all compound expressions in our language should be delimited by parentheses and look like lists. We can recover this consistency by introducing a special form `quote`, which serves the same purpose as the quotation mark. Thus, we would type `(quote a)` instead of `'a`, and we would type `(quote (a b c))` instead of `'(a b c)`. This is precisely how the interpreter works. The quotation mark is just a single-character abbreviation for wrapping the next complete expression with `quote` to form `(quote ⟨`*`expression`*`⟩)`. This is important because it maintains the principle that any expression seen by the interpreter can be manipulated as a data object. For instance, we could construct the expression `(car '(a b c))`, which is the same as `(car (quote (a b c)))`, by evaluating `(list 'car (list 'quote '(a b c)))`.

```
(car '(a b c))
a
(cdr '(a b c))
(b c)
```

In keeping with this, we can obtain the empty list by evaluating `'()`, and thus dispense with the variable `nil`.

One additional primitive used in manipulating symbols is `eq?`, which takes two symbols as arguments and tests whether they are the same.[33] Using `eq?`, we can implement a useful procedure called `memq`. This takes two arguments, a symbol and a list. If the symbol is not contained in the list (i.e., is not `eq?` 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 symbol:

```
(define (memq item x)
  (cond ((null? x) false)
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

For example, the value of

```
(memq 'apple '(pear banana prune))
```

is false, whereas the value of

```
(memq 'apple '(x (apple sauce) y apple pear))
```

is `(apple pear)`.

Exercise 2.53: What would the interpreter print in response to evaluating each of the following expressions?

[33] We can consider two symbols 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).

```
(list 'a 'b 'c)
(list (list 'george))
(cdr '((x1 x2) (y1 y2)))
(cadr '((x1 x2) (y1 y2)))
(pair? (car '(a short list)))
(memq 'red '((red shoes) (blue socks)))
(memq 'red '(red shoes blue socks))
```

Exercise 2.54: Two lists are said to be `equal?` if they contain equal elements arranged in the same order. For example,

```
(equal? '(this is a list) '(this is a list))
```

is true, but

```
(equal? '(this is a list) '(this (is a) list))
```

is false. To be more precise, we can define `equal?` recursively in terms of the basic `eq?` equality of symbols by saying that `a` and `b` are `equal?` if they are both symbols and the symbols are `eq?`, or if they are both lists such that `(car a)` is `equal?` to `(car b)` and `(cdr a)` is `equal?` to `(cdr b)`. Using this idea, implement `equal?` as a procedure.[34]

Exercise 2.55: Eva Lu Ator types to the interpreter the expression

[34]In practice, programmers use `equal?` to compare lists that contain numbers as well as symbols. Numbers are not considered to be symbols. The question of whether two numerically equal numbers (as tested by `=`) are also `eq?` is highly implementation-dependent. A better definition of `equal?` (such as the one that comes as a primitive in Scheme) would also stipulate that if `a` and `b` are both numbers, then `a` and `b` are `equal?` if they are numerically equal.

```
(car ''abracadabra)
```

To her surprise, the interpreter prints back `quote`. Explain.

2.3.2 Example: Symbolic Differentiation

As an illustration of symbol manipulation and a further illustration of data abstraction, consider the design of a procedure that performs symbolic differentiation of algebraic expressions. We would like the procedure 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 procedure are $ax^2 + bx + c$ and x, the procedure 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:

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

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 procedure 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 procedures to implement the following selectors, constructors, and predicates:

`(variable? e)`	Is `e` a variable?
`(same-variable? v1 v2)`	Are `v1` and `v2` the same variable?

`(sum? e)`	Is `e` a sum?
`(addend e)`	Addend of the sum `e`.
`(augend e)`	Augend of the sum `e`.
`(make-sum a1 a2)`	Construct the sum of `a1` and `a2`.
`(product? e)`	Is `e` a product?
`(multiplier e)`	Multiplier of the product `e`.
`(multiplicand e)`	Multiplicand of the product `e`.
`(make-product m1 m2)`	Construct the product of `m1` and `m2`.

Using these, and the primitive predicate `number?`, which identifies numbers, we can express the differentiation rules as the following procedure:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp) (make-sum (deriv (addend exp) var)
                              (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "unknown expression type: DERIV" exp))))
```

This `deriv` procedure 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 the list `(a * x + b)`. However, one especially straightforward choice is to use the same parenthesized prefix notation that Lisp uses for combinations; that is, to represent $ax + b$ as `(+ (* a x) b)`. Then our data representation for the differentiation problem is as follows:

- The variables are symbols. They are identified by the primitive predicate `symbol?`:

  ```
  (define (variable? x) (symbol? x))
  ```

- Two variables are the same if the symbols representing them are `eq?`:

  ```
  (define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))
  ```

- Sums and products are constructed as lists:

  ```
  (define (make-sum a1 a2) (list '+ a1 a2))
  (define (make-product m1 m2) (list '* m1 m2))
  ```

- A sum is a list whose first element is the symbol `+`:

  ```
  (define (sum? x) (and (pair? x) (eq? (car x) '+)))
  ```

- The addend is the second item of the sum list:

  ```
  (define (addend s) (cadr s))
  ```

- The augend is the third item of the sum list:

  ```
  (define (augend s) (caddr s))
  ```

- A product is a list whose first element is the symbol `*`:

  ```
  (define (product? x) (and (pair? x) (eq? (car x) '*)))
  ```

- The multiplier is the second item of the product list:

  ```
  (define (multiplier p) (cadr p))
  ```

- The multiplicand is the third item of the product list:

  ```
  (define (multiplicand p) (caddr p))
  ```

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 '(+ x 3) 'x)
(+ 1 0)
(deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
   (* (+ (* x 0) (* 1 y))
      (+ x 3)))
```

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.

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list '+ a1 a2))))
```

This uses the procedure =number?, which checks whether an expression is equal to a given number:

```
(define (=number? exp num) (and (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:

```
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

Here is how this version works on our three examples:

```
(deriv '(+ x 3) 'x)
1
(deriv '(* x y) 'x)
y
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

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
>
> $$\frac{d(u^n)}{dx} = nu^{n-1}\frac{du}{dx}$$
>
> by adding a new clause to the `deriv` program and defining appropriate procedures `exponentiation?`, `base`, `exponent`, and `make_exponentiation`. (You may use the symbol `**` 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.

> **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 '(* x y (+ x 3)) 'x)
> ```

Try to do this by changing only the representation for sums and products, without changing the `deriv` procedure 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.

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, such as `(x + (3 * (x + (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 standard algebraic notation, such as `(x + 3 * (x + y + 2))`, which drops unnecessary parentheses and assumes that multiplication is done before addition. Can you design appropriate predicates, selectors, and constructors for this notation such that our derivative program still works?

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`, `element_of_set?`, and `adjoin_set`. `Element_of_set?` is a predicate that determines whether a given element is a member of a set. `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. `Union_set` computes the union of two sets, which is the set containing each element that appears in either argument. `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.[35]

[35]If 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`, `(element_of_set? x (adjoin_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`, `(element_of_set? x (union_set S T))` is

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, element_of_set? is similar to the procedure memq of Section 2.3.1. It uses equal? instead of eq? so that the set elements need not be symbols:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr 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 cons to add the object to the list that represents the set:

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

For intersection_set we can use a recursive strategy. If we know how to form the intersection of set2 and the cdr of set1, we only need to decide whether to include the car of set1 in this. But this depends on whether (car set1) is also in set2. Here is the resulting procedure:

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1) (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

---

equal to (or (element_of_set? x S) (element_of_set? x T)) (informally: "The elements of (union S T) are the elements that are in S or in T").

• For any object x, (element_of_set? x '()) is false (informally: "No object is an element of the empty set").

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 `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, `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, `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 `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.

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 `(2 3 2 1 3 2 2)`. Design procedures `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 procedure for the non-duplicate representation? Are there applications for which you would use this representation in preference to the non-duplicate one?

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 `(1 3 6 10)`.

One advantage of ordering shows up in `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:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr 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 `cdr`-s 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 `cdr` of `set1`. Similarly, if `x2` is less than `x1`, then the intersection is given by the intersection of `set1` with the `cdr` of `set2`. Here is the procedure:

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1 (intersection-set (cdr set1)
                                          (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((< x2 x1)
               (intersection-set set1 (cdr 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 `element_of_set?` show how to take advantage of the ordering to produce a procedure that requires on the average about half as many steps as with the unordered representation.

> **Exercise 2.62:** Give a $\Theta(n)$ implementation of `union_set` for sets represented as ordered lists.

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.

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

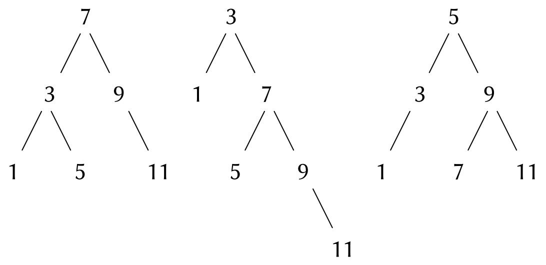


Figure 2.16: Various binary trees that represent the set $\{1, 3, 5, 7, 9, 11\}$.

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 $\Theta(\log n)$.[36] 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 procedures:[37]

[36]Halving 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.

[37]We 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 procedures `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.

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
```

Now we can write the `element_of_set?` procedure using the strategy described above:

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (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 procedure:

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry 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.[38]

Exercise 2.63: Each of the following two procedures converts a binary tree to a list.

```
(define (tree->list-1 tree)
  (if (null? tree)
```

[38]Examples of such structures include *B-trees* and *red-black trees*. There is a large literature on data structures devoted to this problem. See Cormen et al. 1990.

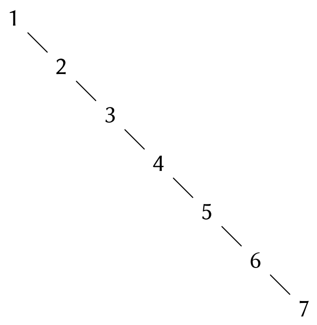


Figure 2.17: Unbalanced tree produced by adjoining 1 through 7 in sequence.

```
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1
                      (right-branch tree))))))
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list
                              (right-branch tree)
                              result-list)))))
  (copy-to-list tree '()))
```

a. Do the two procedures produce the same result for every tree? If not, how do the results differ? What lists do the two procedures produce for the trees in Figure 2.16?

b. Do the two procedures 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?

Exercise 2.64: The following procedure `list_>tree` converts an ordered list to a balanced binary tree. The helper procedure `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 `cons`) whose `car` is the constructed tree and whose `cdr` is the list of elements not included in the tree.

```
(define (list->tree elements)
  (car (partial-tree elements (length elements))))
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result
               (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result
                   (partial-tree
                    (cdr non-left-elts)
                    right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts
                     (cdr right-result)))
                (cons (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_>tree` for the list `(1 3 5 7 9 11)`.

b. What is the order of growth in the number of steps required by `list_>tree` to convert a list of n elements?

Exercise 2.65: Use the results of Exercise 2.63 and Exercise 2.64 to give $\Theta(n)$ implementations of `union_set` and `intersection_set` for sets implemented as (balanced) binary trees.[39]

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

[39]Exercise 2.63 through Exercise 2.65 are due to Paul Hilfinger.

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 procedure that retrieves the key associated with a given record.

Now we represent the data base as a set of records. To locate the record with a given key we use a procedure `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. `Lookup` is implemented in almost the same way as `element_of_set?`. For example, if the set of records is implemented as an unordered list, we could use

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) false)
        ((equal? given-key (key (car set-of-records)))
         (car set-of-records))
        (else (lookup given-key (cdr 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` procedure for the case where the set of records is structured as a binary tree, ordered by the numerical values of the keys.

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 000    C 010    E 100    G 110
B 001    D 011    F 101    H 111
```

With this code, the message

```
BACADAEAFABBAAAGAH
```

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 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

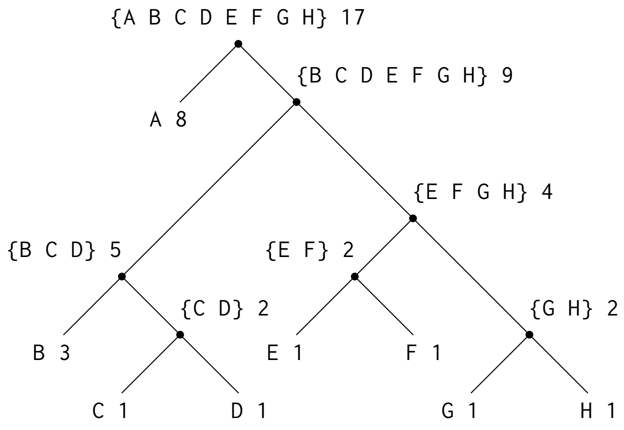


Figure 2.18: A Huffman encoding tree.

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.[40]

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)}
```

[40]See Hamming 1980 for a discussion of the mathematical properties of Huffman codes.

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.

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:

```
(define (make-leaf symbol weight) (list 'leaf symbol weight))
(define (leaf? object) (eq? (car object) 'leaf))
(define (symbol-leaf x) (cadr x))
(define (weight-leaf x) (caddr 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 procedure we defined in Section 2.2.1:

```
(define (make-code-tree left right)
  (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:

```
(define (left-branch  tree) (car  tree))
(define (right-branch tree) (cadr tree))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

The procedures `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 procedures* (procedures that can handle more than one kind of data), which we will have much more to say about in Section 2.4 and Section 2.5.

The decoding procedure

The following procedure implements the decoding algorithm. It takes as arguments a list of zeros and ones, together with a Huffman tree.

```
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
               (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))
```

```
(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit: CHOOSE-BRANCH" bit))))
```

The procedure `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 procedure `choose_branch`.) When it reaches a leaf, it returns the symbol at that leaf as the next symbol in the message by `cons`ing 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 procedure 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` procedure 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.

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
```

The following procedure takes a list of symbol-frequency pairs such as `((A 4) (B 2) (C 1) (D 1))` and constructs an initial ordered set of leaves, ready to be merged according to the Huffman algorithm:

```
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair)     ; symbol
                               (cadr pair))   ; frequency
                    (make-leaf-set (cdr pairs))))))
```

Exercise 2.67: Define an encoding tree and a sample message:

```
(define 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)))))
(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
```

Use the decode procedure to decode the message, and give the result.

Exercise 2.68: The encode procedure takes as arguments a message and a tree and produces the list of bits that gives the encoded message.

```
(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree))))
```

`Encode_symbol` is a procedure, which you must write, 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 procedure 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.

Exercise 2.69: The following procedure 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.

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))
```

`Make_leaf_set` is the procedure given above that transforms the list of pairs into an ordered set of leaves. `Successive_merge` is the procedure you must write, 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 procedure is slightly tricky, but not really complicated. If you find yourself designing a complex procedure, then you are almost certainly doing something wrong. You can take significant advantage of the fact that we are using an ordered set representation.)

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    GET 2    SHA 3    WAH 1
BOOM 1    JOB 2    NA 16    YIP 9
```

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 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
```

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?

Exercise 2.71: Suppose we have a Huffman tree for an alphabet of n symbols, and that the relative frequencies of the symbols are $1, 2, 4, \ldots, 2^{n-1}$. 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?

Exercise 2.72: Consider the encoding procedure 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.

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 procedures that perform rational arithmetic (`add_rat`, `sub_rat`, `mul_rat`, and `div_rat`) from the "higher-level" procedures 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 procedures 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 procedures*—procedures that can operate on data that may be represented in more than one way. Our main technique for building generic procedures 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 procedures 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.

In Section 2.5 we will show how to use type tags and data-directed style to develop a generic arithmetic package. This provides procedures (`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.

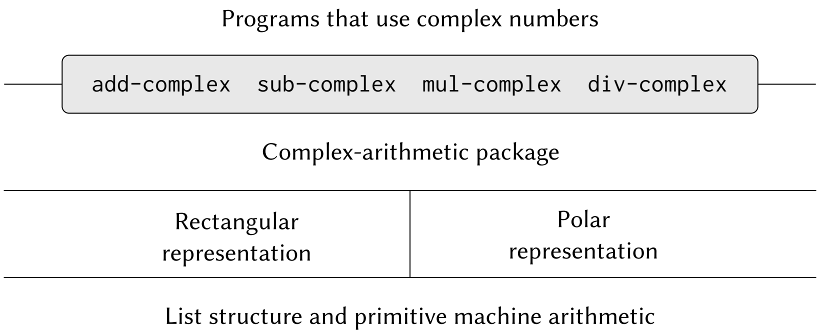

Programs that use complex numbers
add-complex
sub-complex
mul-complex
div-complex
Complex-arithmetic package
Rectangular representation
Polar representation
List structure and primitive machine arithmetic

Figure 2.19: Data-abstraction barriers in the complex-number system.

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).[41] 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

[41]In 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.

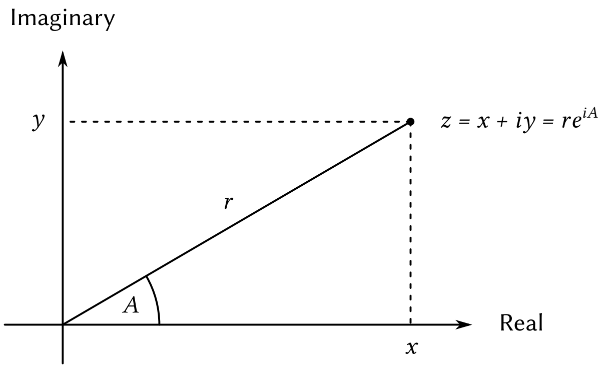


Figure 2.20: Complex numbers as points in the plane.

"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 the real part of a complex number that is specified by polar coordinates.

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 procedures 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 procedures 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:

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (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

$$\begin{aligned} x &= r\cos A, & \qquad r &= \sqrt{x^2+y^2}, \\ y &= r\sin A, & \qquad A &= \arctan(y,x), \end{aligned}$$

which relate the real and imaginary parts (x, y) to the magnitude and the angle (r, A).[42] Ben's representation is therefore given by the following selectors and constructors:

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z))
           (square (imag-part z)))))
(define (angle z)
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (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:

```
(define (real-part z) (* (magnitude z) (cos (angle z))))
(define (imag-part z) (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (make-from-mag-ang r a) (cons 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.

[42]The arctangent function referred to here, computed by Scheme's `atan` procedure, 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.

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 procedures `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 procedure `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:

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
```

```
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum: TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum: CONTENTS" datum)))
```

Using these procedures, we can define predicates `rectangular?` and `polar?`, which recognize rectangular and polar numbers, respectively:

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z) (eq? (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 procedures do not conflict. One way to do this is for Ben to append the suffix `rectangular` to the name of each of his representation procedures and for Alyssa to append `polar` to the names of hers. Here is Ben's revised rectangular representation from Section 2.4.1:

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
           (square (imag-part-rectangular z)))))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))
```

```
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))
```

and here is Alyssa's revised polar representation:

```
(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
```

Each generic selector is implemented as a procedure that checks the tag of its argument and calls the appropriate procedure 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 procedure as required:

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type: REAL-PART" z))))
```

```
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type: IMAG-PART" z))))
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type: MAGNITUDE" z))))
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Unknown type: ANGLE" z))))
```

To implement the complex-number arithmetic operations, we can use the same procedures `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 procedure `add_complex` is still

```
(define (add-complex z1 z2)
  (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:

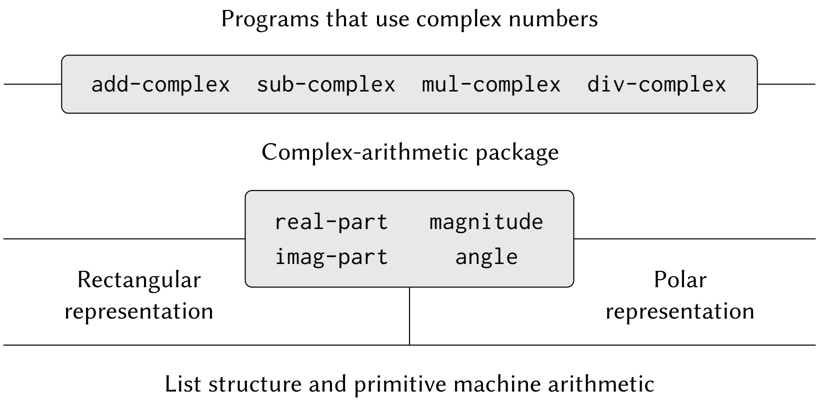

Programs that use complex numbers
add-complex
sub-complex
mul-complex
div-complex
Complex-arithmetic package
real-part
magnitude
imag-part
angle
Rectangular representation
Polar representation
List structure and primitive machine arithmetic

Figure 2.21: Structure of the generic complex-arithmetic system.

```
(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))
(define (make-from-mag-ang r a)
  (make-from-mag-ang-polar r a))
```

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 procedures 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 procedures. 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 procedure 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 procedures (`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 procedures 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 procedures in the entire system have the same name. This is why Ben and Alyssa had to change the names of their original procedures 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 procedures must modify those procedures 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 procedures 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 procedures 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 procedure was spread out among the various conditional clauses in the generic interface procedures. But the same information could have been organized in a table, as shown in Figure 2.22.

		Types	
		Polar	Rectangular
Operations	`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.

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 procedures that each perform an explicit dispatch on type. Here we will implement the interface as a single procedure that looks up the combination of the operation name and argument type in the table to find the correct procedure 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 procedures; we need only add new entries to the table.

To implement this plan, assume that we have two procedures, `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 false.

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 procedures, 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 procedure:

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))

  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Notice that the internal procedures here are the same procedures 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 procedure definitions are internal to the installation procedure, Ben needn't worry about name conflicts with other procedures outside the rectangular package. To interface these to the rest of the system, Ben installs his `real_part` procedure under the operation name `real_part` and the type `(rectangular)`, and similarly for the other selectors.[43] The interface also defines the constructors to be used by the external system.[44] These are identical to Ben's internally defined constructors, except that they attach the tag.

Alyssa's polar package is analogous:

```
(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z) (* (magnitude z) (cos (angle z))))
  (define (imag-part z) (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
```

[43]We use the list `(rectangular)` rather than the symbol `rectangular` to allow for the possibility of operations with multiple arguments, not all of the same type.

[44]The 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.

```
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Even though Ben and Alyssa both still use their original procedures defined with the same names as each other's (e.g., `real_part`), these definitions are now internal to different procedures (see Section 1.1.8), so there is no name conflict.

The complex-arithmetic selectors access the table by means of a general "operation" procedure called `apply_generic`, which applies a generic operation to some arguments. `Apply_generic` looks in the table under the name of the operation and the types of the arguments and applies the resulting procedure if one is present:[45]

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
```

[45]`Apply_generic` uses the dotted-tail notation described in Exercise 2.20, because different generic operations may take different numbers of arguments. In `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.

`Apply_generic` also uses the primitive procedure `apply`, which takes two arguments, a procedure and a list. `Apply` applies the procedure, using the elements in the list as arguments. For example,

```
(apply + (list 1 2 3 4))
```

returns 10.

```
            "No method for these types: APPLY-GENERIC"
            (list op type-tags))))))
```

Using `apply_generic`, we can define our generic selectors as follows:

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle 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:

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))
(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

Exercise 2.73: Section 2.3.2 described a program that performs symbolic differentiation:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum (make-product
                     (multiplier exp)
                     (deriv (multiplicand exp) var))
                    (make-product
                     (deriv (multiplier exp) var)
                     (multiplicand exp))))
        ⟨more rules can be added here⟩
        (else (error "unknown expression type:
                      DERIV" exp))))
```

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 procedure as

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp))
               (operands exp) var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

a. Explain what was done above. Why can't we assimilate the predicates `number?` and `variable?` into the data-directed dispatch?

b. Write the procedures 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 procedures in the opposite way, so that the dispatch line in `deriv` looked like

   ```
   ((get (operator exp) 'deriv) (operands exp) var)
   ```

   What corresponding changes to the derivative system are required?

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 Scheme, 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` procedure that retrieves a specified employee's record from a specified personnel file. The procedure 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` procedure 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` procedure. This should search all the divisions' files for the record of a given employee and return the record. Assume that this procedure 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?

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 procedure 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 procedure 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

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude) (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op: MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

The corresponding `apply_generic` procedure, 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:[46]

[46]One limitation of this organization is it permits only generic procedures of one argument.

```
(define (apply-generic op arg) (arg op))
```

Note that the value returned by make_from_real_imag is a procedure—the internal dispatch procedure. This is the procedure 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 cons, car, and cdr could be defined with no data objects but only procedures. 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 procedure should be analogous to the make_from_real_imag procedure given above.

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?

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 procedures. 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 procedure `add` that operates on whatever numbers are supplied. `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 procedures (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.

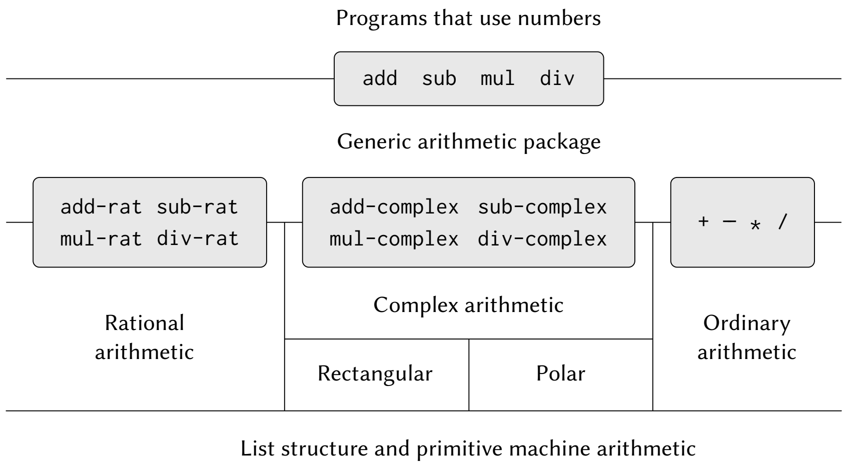


Figure 2.23: Generic arithmetic system.

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 procedure `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 procedure to dispatch to an appropriate package according to the data type of its arguments.

The generic arithmetic procedures are defined as follows:

```
(define (add x y) (apply-generic 'add x y))
```

```
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div 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 `scheme_number`. The arithmetic operations in this package are the primitive arithmetic procedures (so there is no need to define extra procedures to handle the untagged numbers). Since these operations each take two arguments, they are installed in the table keyed by the list `(scheme_number scheme_number)`:

```
(define (install-scheme-number-package)
  (define (tag x) (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)
```

Users of the Scheme-number package will create (tagged) ordinary numbers by means of the procedure:

```
(define (make-scheme-number n)
  ((get 'make 'scheme-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 procedures in the package:

```
(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  (define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  (define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y))))
  (define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (denom x) (numer y))))
  ;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
       (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
       (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational)
       (lambda (x y) (tag (div-rat x y))))
  (put 'make 'rational
       (lambda (n d) (tag (make-rat n d))))
  'done)
(define (make-rational n d)
  ((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 procedures from Section 2.4.1.

```
(define (install-complex-package)
  ;; imported procedures from rectangular and polar packages
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y))
  (define (make-from-mag-ang r a)
    ((get 'make-from-mag-ang 'polar) r a))
  ;; internal procedures
  (define (add-complex z1 z2)
    (make-from-real-imag (+ (real-part z1) (real-part z2))
                         (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
    (make-from-real-imag (- (real-part z1) (real-part z2))
                         (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
    (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                       (+ (angle z1) (angle z2))))
  (define (div-complex z1 z2)
    (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                       (- (angle z1) (angle z2))))
  ;; interface to rest of the system
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2) (tag (add-complex z1 z2))))
  (put 'sub '(complex complex)
       (lambda (z1 z2) (tag (sub-complex z1 z2))))
  (put 'mul '(complex complex)
       (lambda (z1 z2) (tag (mul-complex z1 z2))))
  (put 'div '(complex complex)
       (lambda (z1 z2) (tag (div-complex z1 z2))))
  (put 'make-from-real-imag 'complex
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'complex
       (lambda (r a) (tag (make-from-mag-ang r a))))
  '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 procedures, originally defined in the rectangular and polar packages, are exported to the complex package, and exported from there to the outside world.

```
(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
(define (make-complex-from-mag-ang r a)
  ((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 ap-

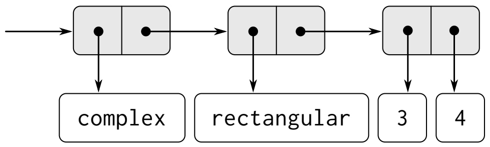


Figure 2.24: Representation of 3 + 4*i* in rectangular form.

propriate package is stripped off (by applying `contents`) and the next level of tag (if any) becomes visible to be used for further dispatching.

In the above packages, we used `add_rat`, `add_complex`, and the other arithmetic procedures exactly as originally written. Once these definitions are internal to different installation procedures, 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)`. 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 '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

Describe in detail why this works. As an example, trace through all the procedures 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 procedure is dispatched to in each case?

Exercise 2.78: The internal procedures in the `scheme_number` package are essentially nothing more than calls to the primitive procedures `+`, `-`, 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 Lisp implementations do have a type system, which they use internally. Primitive predicates such as `symbol?` and `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 Scheme's internal type system. That is to say, the system should work as before except that ordinary numbers should be represented simply as Scheme numbers rather than as pairs whose `car` is the symbol `scheme_number`.

Exercise 2.79: Define a generic equality predicate `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 `=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 procedure 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 procedure for adding complex numbers to ordinary numbers and installs this in the table using the tag `(complex scheme_number)`:[47]

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x) (imag-part z)))
(put 'add '(complex scheme-number)
     (lambda (z x) (tag (add-complex-to-schemenum 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 pack-

[47]We also have to supply an almost identical procedure to handle the types `(scheme_number complex)`.

age of procedures for that type but also the construction and installation of the procedures 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 at 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 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 procedures that transform an object of one type into an equivalent object of another type. Here is a typical coercion procedure, which transforms a given ordinary number to a complex number with that real part and zero imaginary part:

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

We install these coercion procedures in a special coercion table, indexed under the names of the two types:

```
(put-coercion 'scheme-number
              'complex
              scheme-number->complex)
```

(We assume that there are `put_coercion` and `get_coercion` procedures 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_>scheme_number` procedure 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` procedure 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 procedure found in the operation-and-type table. Otherwise, we try coercion. For simplicity, we consider only the case where there are two arguments.[48] We check the coercion table to see if objects of the first type can be coerced to the second type. If so, we

[48]See Exercise 2.82 for generalizations.

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 procedure:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                         (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                         (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types"
                                     (list op type-tags))))))
              (error "No method for these types"
                     (list op type-tags)))))))
```

This coercion scheme has many advantages over the method of defining explicit cross-type operations, as outlined above. Although we still need to write coercion procedures to relate the types (possibly n^2 procedures for a system with n types), we need to write only one procedure for each pair of types rather than a different procedure for each collection of types and each generic operation.[49] What we are counting on here is the fact that the appropriate transformation between types depends only on the types themselves, not on the operation to be applied.

On the other hand, there may be applications for which our coercion scheme is not general enough. Even when neither of the objects to be combined can be converted to the type of the other it may still be possible to perform the operation by converting both objects to a third type. In order to deal with such complexity and still preserve modularity in our programs, it is usually necessary to build systems that take advantage of still further structure in the relations among types, as we discuss next.

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 num-

[49]If we are clever, we can usually get by with fewer than n^2 coercion procedures. For instance, if we know how to convert from type 1 to type 2 and from type 2 to type 3, then we can use this knowledge to convert from type 1 to type 3. This can greatly decrease the number of coercion procedures we need to supply explicitly when we add a new type to the system. If we are willing to build the required amount of sophistication into our system, we can have it search the "graph" of relations among types and automatically generate those coercion procedures that can be inferred from the ones that are supplied explicitly.

$$\begin{tikzcd}
\text{complex} \\
\text{real} \arrow[u] \\
\text{rational} \arrow[u] \\
\text{integer} \arrow[u]
\end{tikzcd}$$

Figure 2.25: A tower of types.

bers (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.

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 procedure `integer_>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` procedure in the following way: For each type, we need to supply a `raise` procedure, 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. (Exercise 2.83 and Exercise 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 procedure 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 $6 + 2i$.)

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,

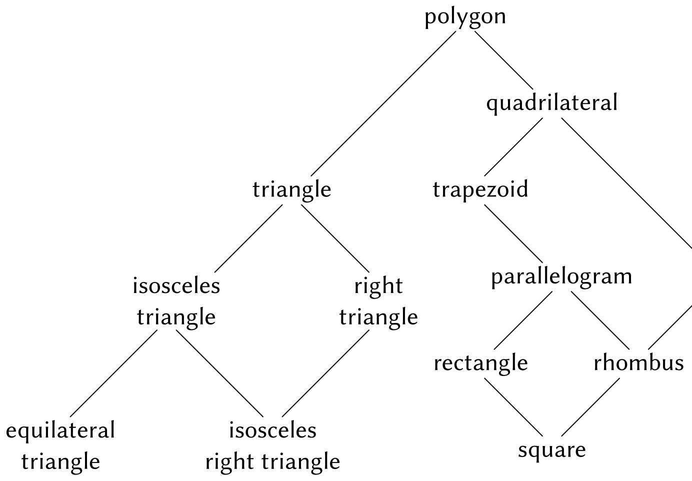


Figure 2.26: Relations among types of geometric figures.

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 procedure 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.[50]

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 procedures in the coercion table to *coerce* arguments of each type to their own type. For example, in addition to the scheme_number_>complex coercion shown above, he would do:

```
(define (scheme-number->scheme-number n) n)
(define (complex->complex z) z)
(put-coercion 'scheme-number
              'scheme-number
              scheme-number->scheme-number)
(put-coercion 'complex 'complex complex->complex)
```

[50]This 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.

a. With Louis's coercion procedures installed, what happens if apply_generic is called with two arguments of type scheme_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:

```
(define (exp x y) (apply-generic 'exp x y))
```

and have put a procedure for exponentiation in the Scheme-number package but not in any other package:

```
;; following added to Scheme-number package
(put 'exp '(scheme-number scheme-number)
     (lambda (x y) (tag (expt x y))))
     ; using primitive expt
```

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 procedure 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` procedure 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 procedure `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` procedure that drops an object as far as possible. You will need to design the various projection operations[51] 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.

[51] A 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*).[52] 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,

$$5x^2 + 3x + 7$$

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.[53] We can finesse these ques-

[52]On 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.

[53]For 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

tions 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 `same_variable?` procedure of Section 2.3.2 to compare variables. The following procedures define addition and multiplication of polys:

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var: ADD-POLY" (list p1 p2))))
(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (mul-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var: MUL-POLY" (list p1 p2))))
```

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.

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 procedure for the polynomial package, similar to the ones in Section 2.5.1:

```
(define (install-polynomial-package)
  ;; internal procedures
  ;; representation of poly
  (define (make-poly variable term-list) (cons variable term-list))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  ⟨procedures same-variable? and variable? from section 2.3.2⟩
  ;; representation of terms and term lists
  ⟨procedures adjoin-term ... coeff from text below⟩
  (define (add-poly p1 p2) ...)
  ⟨procedures used by add-poly⟩
  (define (mul-poly p1 p2) ...)
  ⟨procedures used by mul-poly⟩
  ;; interface to rest of the system
  (define (tag p) (attach-tag 'polynomial p))
  (put 'add '(polynomial polynomial)
       (lambda (p1 p2) (tag (add-poly p1 p2))))
  (put 'mul '(polynomial polynomial)
       (lambda (p1 p2) (tag (mul-poly p1 p2))))
  (put 'make 'polynomial
       (lambda (var terms) (tag (make-poly var terms))))
  '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 `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 procedure that constructs the term list for the sum of two polynomials:[54]

```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1))
               (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term
                   t1 (add-terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
```

[54]This 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`.

```
                  (adjoin-term
                   t2 (add-terms L1 (rest-terms L2))))
                 (else
                  (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 procedure add to add together the coefficients of the terms being combined. This has powerful consequences, 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:

```
(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2))))
(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (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 procedures `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

$$[3x^2 + (2 + 3i)x + 7] \cdot \left[x^4 + \frac{2}{3}x^2 + (5 + 3i)\right].$$

Because we installed the polynomial addition and multiplication procedures `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

$$\left[(y + 1)x^2 + (y^2 + 1)x + (y - 1)\right] \cdot \left[(y - 2)x + (y^3 + 7)\right].$$

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.[55]

[55]To 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

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 procedures `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 `(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,

---

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.

decremented by 1.[56] 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 `((100 1) (2 2) (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-order to lowest-order term. Once we have made this decision, implementing the selectors and constructors for terms and term lists is straightforward:[57]

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
(define (make-term order coeff) (list order coeff))
```

[56]In 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 `car` is the symbol `scheme_number`.

[57]Although we are assuming that term lists are ordered, we have implemented `adjoin_term` to simply `cons` the new term onto the existing term list. We can get away with this so long as we guarantee that the procedures (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).

```
(define (order term) (car term))
(define (coeff term) (cadr term))
```

where =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 procedure:

```
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

Exercise 2.87: Install =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: 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 procedures 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` procedure on the model of `add_poly` and `mul_poly`. The procedure 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.

```
(define (div-terms L1 L2)
  (if (empty-termlist? L1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) L1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (- (order t1) (order t2))))
              (let ((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:
>
> ```
> (define p1 (make-polynomial 'x '((2 1) (0 1))))
> (define p2 (make-polynomial 'x '((3 1) (0 1))))
> (define rf (make-rational p2 p1))
> ```
>
> Now add `rf` to itself, using `add`. You will observe that this addition procedure 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.[58] The integer version is

```
(define (gcd a b)
  (if (= 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:

```
(define (gcd-terms a b)
  (if (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.

[58]The 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.

Exercise 2.94: Using `div_terms`, implement the procedure `remainder_terms` and use this to define `gcd_terms` as above. Now write a procedure `gcd_poly` that computes the polynomial GCD of two polys. (The procedure 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

```
(define p1 (make-polynomial
            'x '((4 1) (3 -1) (2 -2) (1 2))))
(define p2 (make-polynomial 'x '((3 1) (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

$$\begin{aligned} P_1 &: \quad x^2 - 2x + 1, \\ P_2 &: \quad 11x^2 + 7, \\ P_3 &: \quad 13x + 5. \end{aligned}$$

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.[59] To understand what is

[59]In an implementation like MIT Scheme, this produces a polynomial that is indeed a divisor of Q_1 and Q_2, but with rational coefficients. In many other Scheme systems, in which division of integers can produce limited-precision decimal numbers, we may fail to get a valid divisor.

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:

a. Implement the procedure `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 procedure `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 procedure 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 procedure analogous to reduce_terms that does what the original make_rat did for integers:

```
(define (reduce-integers n d)
  (let ((g (gcd n d)))
    (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 scheme_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:

```
(define  p1 (make-polynomial 'x '((1 1) (0  1))))
(define  p2 (make-polynomial 'x '((3 1) (0 -1))))
(define  p3 (make-polynomial 'x '((1 1))))
(define  p4 (make-polynomial 'x '((2 1) (0 -1))))
(define rf1 (make-rational p1 p2))
(define 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.[60]

[60]One 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 (Zippel 1993) describes this method, together with other ways to compute polynomial GCDs.

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. (Onl)

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. (Online)

Backus, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8): 613-641. (Online)

Baker, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4): 280-293. (Online)

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*. (Online)

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. (Online)

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. (Online)

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. (Online)

Dijkstra, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43-112. (Online)

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. (Online)

Doyle, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12: 231-272. (Online)

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. (Online)

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. (Online)

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. (Online)

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. (Online)

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): 396-404. (Online)

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). (Online)

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. (Online)

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. (Online) (2002 version)

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. (Online)

Hewitt, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3): 323-364. (Online)

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. (Online)

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.

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.

Kohlbecker, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University. (Online)

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.

Kowalski, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh. (Online)

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. (Online)

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. (Online)

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. (Online)

Liskov, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. IEEE *Transactions on Software Engineering* 1(1): 7-19. (Online)

McAllester, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory. (Online)

McAllester, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory. (Online)

McCarthy, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4): 184-195. (Online)

McCarthy, John. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland. (Online)

McCarthy, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages*. (Online)

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. (Online)

McDermott, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory. (Online)

Miller, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3): 300-317. (Online)

Miller, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory. (Online)

Moon, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science. (Online)

Moon, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory. (Online)

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. (Online)

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. (Online)

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. (Online)

Rees, Jonathan, and William Clinger (eds). 1991. The revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3). (Online)

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. (Online)

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.

Spafford, Eugene H. 1989. The Internet Worm: Crisis and aftermath.

Communications of the ACM 32(6): 678-688. (Online)

Steele, Guy Lewis, Jr. 1977. Debunking the "expensive procedure call" myth. In *Proceedings of the National Conference of the ACM*, pp. 153-62. (Online)

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. (Online)

Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory. (Online)

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. (Online)

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. (Online)

Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierachical descriptions. *AI Journal* 14: 1-39. (Online)

Sussman, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257: 256-262. (Online)

Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory. (Online)

Sutherland, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory. (Onl.)

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. (Online)

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. (Online)

Winston, Patrick. 1992. *Artificial Intelligence*. 3rd edition. Reading, MA: Addison-Wesley.

Zabih, Ramin, David McAllester, and David Chapman. 1987. Nondeterministic Lisp with dependency-directed backtracking. *AAAI-87*, pp. 59-64. (Online)

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.

List of Exercises

Chapter 1

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
1.11	1.12	1.13	1.14	1.15	1.16	1.17	1.18	1.19	1.20
1.21	1.22	1.23	1.24	1.25	1.26	1.27	1.28	1.29	1.30
1.31	1.32	1.33	1.34	1.35	1.36	1.37	1.38	1.39	1.40
1.41	1.42	1.43	1.44	1.45	1.46				

Chapter 2

2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
2.11	2.12	2.13	2.14	2.15	2.16	2.17	2.18	2.19	2.20
2.21	2.22	2.23	2.24	2.25	2.26	2.27	2.28	2.29	2.30
2.31	2.32	2.33	2.34	2.35	2.36	2.37	2.38	2.39	2.40
2.41	2.42	2.43	2.44	2.45	2.46	2.47	2.48	2.49	2.50
2.51	2.52	2.53	2.54	2.55	2.56	2.57	2.58	2.59	2.60
2.61	2.62	2.63	2.64	2.65	2.66	2.67	2.68	2.69	2.70
2.71	2.72	2.73	2.74	2.75	2.76	2.77	2.78	2.79	2.80
2.81	2.82	2.83	2.84	2.85	2.86	2.87	2.88	2.89	2.90
2.91	2.92	2.93	2.94	2.95	2.96	2.97			

Chapter 3

3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.11	3.12	3.13	3.14	3.15	3.16	3.17	3.18	3.19	3.20
3.21	3.22	3.23	3.24	3.25	3.26	3.27	3.28	3.29	3.30
3.31	3.32	3.33	3.34	3.35	3.36	3.37	3.38	3.39	3.40
3.41	3.42	3.43	3.44	3.45	3.46	3.47	3.48	3.49	3.50
3.51	3.52	3.53	3.54	3.55	3.56	3.57	3.58	3.59	3.60
3.61	3.62	3.63	3.64	3.65	3.66	3.67	3.68	3.69	3.70
3.71	3.72	3.73	3.74	3.75	3.76	3.77	3.78	3.79	3.80
3.81	3.82								

Chapter 4

4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10
4.11	4.12	4.13	4.14	4.15	4.16	4.17	4.18	4.19	4.20
4.21	4.22	4.23	4.24	4.25	4.26	4.27	4.28	4.29	4.30
4.31	4.32	4.33	4.34	4.35	4.36	4.37	4.38	4.39	4.40
4.41	4.42	4.43	4.44	4.45	4.46	4.47	4.48	4.49	4.50
4.51	4.52	4.53	4.54	4.55	4.56	4.57	4.58	4.59	4.60
4.61	4.62	4.63	4.64	4.65	4.66	4.67	4.68	4.69	4.70
4.71	4.72	4.73	4.74	4.75	4.76	4.77	4.78	4.79	

Chapter 5

5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10
5.11	5.12	5.13	5.14	5.15	5.16	5.17	5.18	5.19	5.20
5.21	5.22	5.23	5.24	5.25	5.26	5.27	5.28	5.29	5.30
5.31	5.32	5.33	5.34	5.35	5.36	5.37	5.38	5.39	5.40
5.41	5.42	5.43	5.44	5.45	5.46	5.47	5.48	5.49	5.50
5.51	5.52								

List of Figures

Chapter 5

5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 5.10
5.11 5.12 5.13 5.14 5.15 5.16 5.17 5.18

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*)

Colophon

ON THE COVER PAGE is Agostino Ramelli's bookwheel mechanism from 1588. It could be seen as an early hypertext navigation aid. This image of the engraving is hosted by J. E. Johnson of New Gottland.

The typefaces are Linux Libertine for body text and Linux Biolinum for headings, both by Philipp H. Poll. Typewriter face is Inconsolata created by Raph Levien and supplemented by Dimosthenis Kaponis and Takashi Tanigawa in the form of Inconsolata LGC.

Graphic design and typography are done by Andres Raba. Texinfo source is converted to LaTeX by a Perl script and compiled to PDF by XeLaTeX. Diagrams are drawn with Inkscape.