Structure and Interpretation of Computer Programs — JavaScript Adaptation

Harold Abelson and Gerald Jay Sussman
with Julie Sussman
— *authors*

Martin Henz and Tobias Wrigstad
with Liu Hang, Feng Piaopiao, Jolyn Tan and Chan Ger Hean
— *adapters to JavaScript*

Contents

Foreword

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

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

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

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

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

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

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

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

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

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

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

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

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

— Alan J. Perlis, New Haven, Connecticut

Prefaces

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

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

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

The language Scheme has been designed as a sublanguage of Lisp with its use in education as a central design objective. The language JavaScript, on the other hand, was not designed with the needs of learners in mind. This makes it difficult to use JavaScript in a course, even if one imposes constraints on the language features to be covered. The reason is that students with

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

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

— Martin Henz

> 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 , we have students use the register-machine simulator but we do not cover its implementation , and we give only a cursory overview of the compiler . Even so, this is still an intense course. Some instructors may wish to cover only the first three or four chapters, leaving the other material for subsequent courses.

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

— Harold Abelson and Gerald Jay Sussman

> A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists. Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.
>
> — Marvin Minsky, 'Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily-Formulated Ideas'

'The Structure and Interpretation of Computer Programs' is the entry-level subject in computer science at the Massachusetts Institute of Technology. It is required of all students at MIT who major in electrical engineering or in computer science, as one-fourth of the 'common core curriculum,' which also includes two subjects on circuits and linear systems and a subject on the design of digital systems. We have been involved in the development of this subject since 1978, and we have taught this material in its present form since the fall of 1980 to between 600 and 700 students each year. Most of these students have had little or no prior formal training in computation, although many have played with computers a bit and a few have had extensive programming or hardware-design experience.

Our design of this introductory computer-science subject reflects two major concerns. First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about

methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems.

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

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

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

In teaching our material we use a dialect of the programming language Lisp. We never formally teach the language, because we don't have to. We just use it, and students pick it up in a few days. This is one great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. All of the formal properties can be covered in an hour, like the rules of chess. After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues—figuring out what we want to compute, how we will decompose problems into manageable parts, and how we will work on the parts. Another advantage of Lisp is that it supports (but does not enforce) more of the large-scale strategies for modular decomposition of programs than any other language we know. We can make procedural and data abstractions, we can use higher-order functions to capture common patterns of usage, we can model local state using assignment and data

mutation, we can link parts of a program with streams and delayed evaluation, and we can easily implement embedded languages. All of this is embedded in an interactive environment with excellent support for incremental program design, construction, testing, and debugging. We thank all the generations of Lisp wizards, starting with John McCarthy, who have fashioned a fine tool of unprecedented power and elegance.

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

— Harold Abelson and Gerald Jay Sussman

Acknowledgments

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

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

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

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

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

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

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

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

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

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

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

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

It is hard to enumerate all the people who have made technical contributions to the development of the Scheme systems we use for instructional purposes. In addition to Guy Steele, principal wizards have included Chris Hanson, Joe Bowbeer, Jim Miller, Guillermo Rozas, and Stephen Adams. Others who have put in significant time are Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, John Lamping, Gwyn Osnos, Tracy Larrabee, George 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 R4RS, and Chris Haynes, David Bartley, Chris Hanson, and Jim Miller, who prepared the IEEE standard.

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

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

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

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

— Harold Abelson and Gerald Jay Sussman

Chapter 1

Building Abstractions with Functions

> The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.
>
> — John Locke, An Essay Concerning Human Understanding (1690)

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

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

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

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

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

Programming in JavaScript

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

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

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

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

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

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

Chapter 2

Building Abstractions with Data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.1 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 2.1 shows a standard way to visualize a pair—in this case, the pair formed by `pair(1,2)`.

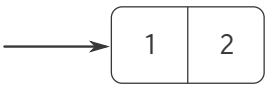
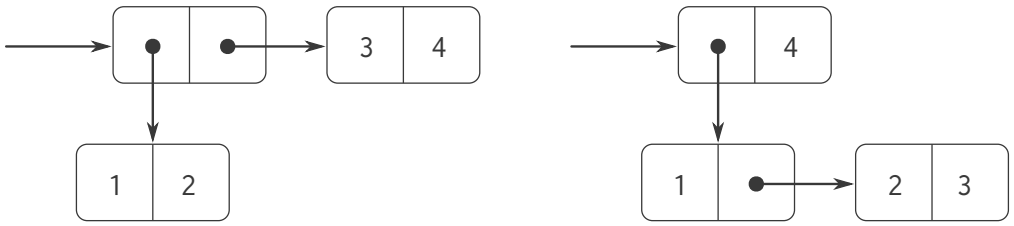


Figure 2.1: Box-and-pointer representation of `pair(1,2)`.

In this representation, which is called *box-and-pointer notation*, each compound object is shown as a *pointer* to a box. The box for a pair has two parts, the left part containing the head of the pair and the right part containing the tail.[2]

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

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




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

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

Figure 2.2: Two ways to combine 1, 2, 3, and 4 using pairs.

The ability to create pairs whose elements are pairs is the essence of list structure's importance as a representational tool. We refer to this ability as the *closure property* of `pair`. In general, an operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation.[3] Closure is the key to power in any means of combination because it permits us to create *hierarchical* structures—structures made up of parts, which themselves are made up of parts, and so on.

From the outset of chapter 1, we've made essential use of closure in dealing with functions, because all but the very simplest programs rely on the fact that the elements of a combination can themselves be combinations. In this section, we take up the consequences of closure for compound data. We describe some conventional techniques for using pairs to represent sequences and trees, and we exhibit a graphics language that illustrates closure in a vivid way.[4]

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

[4]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.'

2.1.1 Representing Sequences

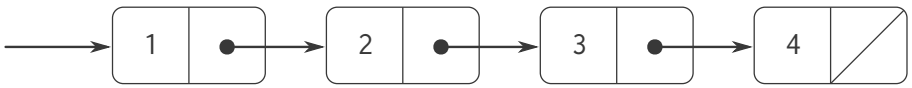


Figure 2.3: The sequence 1, 2, 3, 4 represented as a chain of pairs.

One of the useful structures we can build with pairs is a *sequence*—an ordered collection of data objects. There are, of course, many ways to represent sequences in terms of pairs. One particularly straightforward representation is illustrated in figure 2.3, where the sequence 1, 2, 3, 4 is represented as a chain of pairs. The head of each pair is the corresponding item in the chain, and the `tail` of the pair is the next pair in the chain. The `tail` of the final pair signals the end of the sequence by pointing to a distinguished value that is not a pair, represented in box-and-pointer diagrams as a diagonal line and in programs as the value of JavaScript's value `null` . The entire sequence is constructed by nested `pair` operations:

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

Such a sequence of pairs is called a *list*, and our JavaScript environment provides a primitive called `list` to help in constructing lists.[5]

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

`list(`a_1`, `a_2`, ..., `a_n`)`

is equivalent to

`pair(`a_1`, pair(`a_2`, pair(..., pair(`a_n`, null)...)))`

Our interpreter prints pairs using a textual representation of box-and-pointer diagrams. The result of `pair(1, 2)` is printed as `[1, 2]`, and the data object in figure 2.3 is printed as `[1, [2, [3, [4, null]]]]`:

```
const one_through_four = list(1, 2, 3, 4);
```

We can think of head as selecting the first item in the list, and of `tail` as selecting the sublist consisting of all but the first item. Nested applications of head and `tail` can be used to extract the second, third, and subsequent items in the list. The constructor `pair` makes a list like the original one, but with an additional item at the beginning.

```
head(one_through_four);
// result: 1
```

[5] In this book, we use *list* to mean a chain of pairs terminated by the end-of-list marker. In contrast, the term *list structure* refers to any data structure made out of pairs, not just to lists.

```
tail(one_through_four);
// result: [2, [3, [4, null]]]
```

```
head(tail(one_through_four));
// result: 2
```

```
pair(10, one_through_four);
// result: [10, [1, [2, [3, [4, null]]]]]
```

```
pair(5, one_through_four);
// result: [5, [1, [2, [3, [4, null]]]]]
```

The value `null`, used to terminate the chain of pairs, can be thought of as a sequence of no elements, the *empty list*.

List operations

The use of pairs to represent sequences of elements as lists is accompanied by conventional programming techniques for manipulating lists by successively '`tail`ing down' the lists. For example, the function `list_ref` takes as arguments a list and a number n and returns the nth item of the list. It is customary to number the elements of the list beginning with 0. The method for computing `list_ref` is the following:

- For $n = 0$, `list_ref` should return the `head` of the list.
- Otherwise, `list_ref` should return the $(n - 1)$st item of the `tail` of the list.

```
function list_ref(items, n) {
    return n === 0
           ? head(items)
           : list_ref(tail(items), n - 1);
}
```

```
const squares = list(1, 4, 9, 16, 25);
list_ref(squares, 3);
// result: 16
```

Often we `tail` down the whole list. To aid in this, our JavaScript environment includes a predicate `is_null`, which tests whether its argument is the empty list. The function `length`, which returns the number of items in a list, illustrates this typical pattern of use:

```
function length(items) {
    return is_null(items)
           ? 0
           : 1 + length(tail(items));
}
```

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

- The length of any list is 1 plus the length of the tail of the list.

This is applied successively until we reach the base case:

- The length of the empty list is 0.

We could also compute length in an iterative style:

```
function length(items) {
    function length_iter(a, count) {
        return is_null(a)
               ? count
               : length_iter(tail(a), count + 1);
    }
    return length_iter(items, 0);
}
```

Another conventional programming technique is to 'pair up' an answer list while tailing down a list, as in the function append, which takes two lists as arguments and combines their elements to make a new list:

```
append(squares, odds);
```

```
// returns: [1, [4, [9, [16, [25, [1, [3, [5, [7, null]]]]]]]]]
```

```
append(odds, squares);
```

The function append is also implemented using a recursive plan. To append lists list1 and list2, do the following:

- If list1 is the empty list, then the result is just list2.
- Otherwise, append the tail of list1 and list2, and pair the head of list1 onto the result:

```
function append(list1, list2) {
    return is_null(list1)
           ? list2
           : pair(head(list1), append(tail(list1), list2));
}
```

Exercise 2.1

Define a function last_pair that returns the list that contains only the last element of a given (nonempty) list:

```
last_pair(list(23, 72, 149, 34));
// result: [34, null]
```

Solution

Exercise 2.2

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

```
reverse(list(1, 4, 9, 16, 25));
// result: [25, [16, [9, [4, [1, null]]]]]
```

Solution

Exercise 2.3

Consider the change-counting program of section ??. It would be nice to be able to easily change the currency used by the program, so that we could compute the number of ways to change a British pound, for example. As the program is written, the knowledge of the currency is distributed partly into the function `first_denomination` and partly into the function count_change (which knows that there are five kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

We want to rewrite the function cc so that its second argument is a list of the values of the coins to use rather than an integer specifying which coins to use. We could then have lists that defined each kind of currency:

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

We could then call cc as follows:

```
cc(100, us_coins);
```

To do this will require changing the program cc somewhat. It will still have the same form, but it will access its second argument differently, as follows:

```
function cc(amount, coin_values) {
    return amount === 0
           ? 1
           : amount < 0 || no_more(coin_values)
             ? 0
             : cc(amount,
                  except_first_denomination(coin_values))
               +
```

```
                cc(amount - first_denomination(coin_values),
                   coin_values);
}
```

Define the functions `first_denomination`, `except_first_denomination`, and `no_more` in terms of primitive operations on list structures. Does the order of the list `coin_values` affect the answer produced by `cc`? Why or why not? Solution

Exercise 2.4

In the presence of higher-order functions, it is not strictly necessary for functions to have multiple parameters; one would suffice.[6] If we have a function such as `plus` that naturally requires two parameters, we could write a variant of the function to which we pass the arguments one at at time. An application of the variant to the first argument could return a function that we can then apply to the second argument, and so on. This practice—called *currying* and named after the American mathematician and logician Haskell Brooks Curry—is quite common in programming languages such as Haskell (the reader might venture a guess after whom this programming language is named) and Ocaml. In JavaScript, a curried version of `plus` looks as follows.

```
function plus_curried(x) {
    return y => x + y;
}
```

Write a function `brooks`, that takes a curried function as first argument and as second argument a list of arguments to which the curried function is then applied, one by one, in the given order. For example, the following application of `brooks` should have the same effect as the call `plus_curried(3)(4)` above.

```
brooks(plus_curried, list(3, 4));
```

While we are at it, we might as well curry the function `brooks`! Write a function `brooks_curried` that can be applied as follows, to yield the same result 7:

```
brooks_curried(list(plus_curried, 3, 4));
```

With this function `brooks_curried` what are the results of evaluating the following two statements?

```
brooks_curried(list(brooks_curried,
                    list(plus_curried, 3, 4)));
```

[6] Exercise 2.20 of the original textbook deals with Scheme operators that take variable numbers of arguments. This concept exists in JavaScript, but plays a less prominent role. The textbook adaptors decided to sneak in currying on this occasion.

```
brooks_curried(list(brooks_curried,
                    list(brooks_curried,
                         list(plus_curried, 3, 4))));
```

Solution

Mapping over lists

One extremely useful operation is to apply some transformation to each element in a list and generate the list of results. For instance, the following function scales each number in a list by a given factor:

```
function scale_list(items, factor) {
    return is_null(items)
           ? null
           : pair(head(items) * factor,
                  scale_list(tail(items), factor));
}
```

We can abstract this general idea and capture it as a common pattern expressed as a higher-order function, just as in section **??**. The higher-order function here is called `map`. The function `map` takes as arguments a function of one argument and a list, and returns a list of the results produced by applying the function to each element in the list:

```
function map(fun, items) {
    return is_null(items)
           ? null
           : pair(fun(head(items)),
                  map(fun, tail(items)));
}
```

```
map(abs, list(-10, 2.5, -11.6, 17));
```

```
map(x => x * x, list(1, 2, 3, 4));
```

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

```
function scale_list(items, factor) {
    return map(x => x * factor, items);
}
```

The function `map` is an important construct, not only because it captures a common pattern, but because it establishes a higher level of abstraction in dealing with lists. In the original definition of `scale_list`, the recursive structure of the program draws attention to the element-by-element processing of the list. Defining `scale_list` in terms of `map` suppresses that level of detail and emphasizes that scaling transforms a list of elements to a list of results. The difference between the two definitions is not that the computer is performing a different process (it

isn't) but that we think about the process differently. In effect, `map` helps establish an abstraction barrier that isolates the implementation of functions that transform lists from the details of how the elements of the list are extracted and combined. Like the barriers shown in figure **??**, 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 2.1.3 expands on this use of sequences as a framework for organizing programs.

Exercise 2.5

The function `square_list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
square_list(list(1, 2, 3, 4));
// returns: [1, [4, [9, [16, null]]]]
```

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

```
function square_list(items) {
    return is_null(items)
           ? null
           : pair(??, ??);
}
```

```
function square_list(items) {
   return map(??, ??);
}
```

Solution

Exercise 2.6

Louis Reasoner tries to rewrite the first `square_list` function of exercise 2.5 so that it evolves an iterative process:

```
function square_list(items) {
    function iter(things, answer) {
        return is_null(things)
               ? answer
               : iter(tail(things),
                      pair(square(head(things)),
                           answer));
    }
    return iter(items, null);
}
```

Unfortunately, defining `square_list` this way produces the answer list in the reverse order of the one desired. Why? Louis then tries to fix his bug by interchanging the arguments to `pair`:

```
function square_list(items) {
   function iter(things, answer) {
       return is_null(things)
              ? answer
              : iter(tail(things),
                     pair(answer,
                          square(head(things))));
   }
   return iter(items, null);
}
```

This doesn't work either. Explain. Solution

Exercise 2.7

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

```
for_each(x => display(x),
         list(57, 321, 88));
```

The value returned by the call to `for_each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for_each`. Solution

2.1.2 Hierarchical Structures

The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences. For example, we can regard the object `[[1, [2, null]], [3, [` constructed by

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

as a list of three items, the first of which is itself a list, `[1, [2, null]]`. Figure 2.4 shows the representation of this structure in terms of pairs.

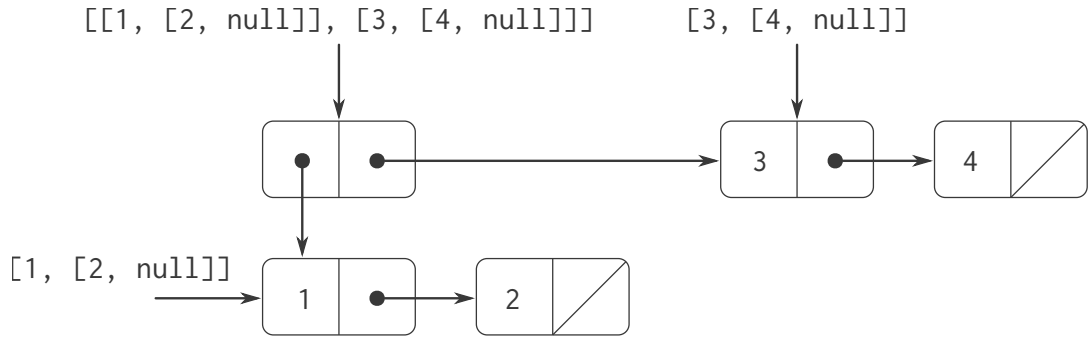


Figure 2.4: Structure formed by `pair(list(1, 2), list(3, 4))`.

Another way to think of sequences whose elements are sequences is as *trees*. The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees. Figure 2.5 shows the structure in Figure 2.4 viewed as a tree.

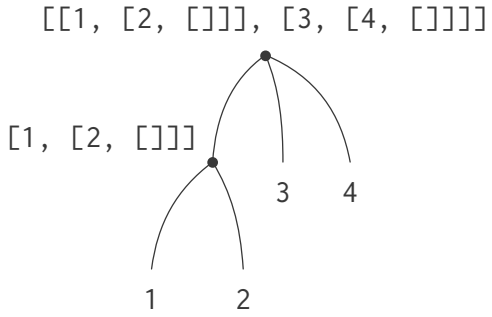


Figure 2.5: The list structure in Figure 2.4 viewed as a tree.

Recursion is a natural tool for dealing with tree structures, since we can often reduce operations on trees to operations on their branches, which reduce in turn to operations on the branches of the branches, and so on, until we reach the leaves of the tree. As an example, compare the `length` function of section 2.1.1 with the `count_leaves` function, which returns the total number of leaves of a tree:

```
const x = pair(pair(1, pair(2,null)), pair(3, pair(4,null)));
```

```
length(x);
// 3
```

```
count_leaves(x);
// 4
```

```
list(x, x);
// [[[[1, [2, null]], [3, [4, null]]],
//  [[[1, [2, null]], [3, [4, null]]], null]]
```

```
length(list(x, x));
// 2
```

```
count_leaves(list(x, x));
// 8
```

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

- The `length` of a list `x` is 1 plus the `length` of the `tail` of `x`.
- The `length` of the empty list is 0.

The function `count_leaves` is similar. The value for the empty list is the same:

- `count_leaves` of the empty list is 0.

But in the reduction step, where we strip off the `head` of the list, we must take into account that the `head` may itself be a tree whose leaves we need to count. Thus, the appropriate reduction step is

- `count_leaves` of a tree `x` is `count_leaves` of the `head` of `x` plus `count_leaves` of the `tail` of `x`.

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

- `count_leaves` of a leaf is 1.

To aid in writing recursive functions on trees, our JavaScript environment provides the primitive predicate `is_pair`, which tests whether its argument is a pair. Here is the complete function:

```
function count_leaves(x) {
    return is_null(x)
           ? 0
           : ! is_pair(x)
             ? 1
             : count_leaves(head(x)) +
               count_leaves(tail(x));
}
```

Exercise 2.8

Suppose we evaluate the expression `list(1, list(2, list(3, 4)))`. Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in Figure 2.5). Solution

Exercise 2.9

Give combinations of `heads` and `tails` that will pick 7 from each of the following lists, given as printed by our interpreter:

```
[1, [3, [[5, [7, null]], [9, null]]]]
```

```
[[7, null], null]
```

```
[1,
 [
  [2,
   [
    [3,
     [
      [4,
       [
        [5,
         [
          [6,
           [7,
            null
           ]
          ],
          null
         ]
        ],
        null
       ]
      ],
      null
     ]
    ],
    null
   ]
  ],
  null
 ]
]
```

Solution

Exercise 2.10

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

```
const x = list(1, 2, 3);

const y = list(4, 5, 6);
```

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

```
append(x, y);
```

```
pair(x, y);
```

```
list(x, y);
```

Solution

Exercise 2.11

Modify your `reverse` function of exercise 2.2 to produce a `deep_reverse` function that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

```
const x = list(list(1, 2), list(3, 4));
```

```
x;
// [[1, [2, null]], [[3, [4, null]], null]]
```

```
reverse(x);
// [[3, [4, null]], [[1, [2, null]], null]]
```

```
deep_reverse(x);
// [[4, [3, null]], [[2, [1, null]], null]]
```

Solution

Exercise 2.12

Write a function `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
const x = list(list(1, 2), list(3, 4));
```

```
fringe(x);
// [1, [2, [3, [4, null]]]]
```

```
fringe(list(x, x));
// [1, [2, [3, [4, [1, [2, [3, [4, null]]]]]]]]
```

Solution

Exercise 2.13

A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

```
function make_mobile(left, right) {
   return list(left, right);
}
```

A branch is constructed from a `length` (which must be a number) together with a `structure`, which may be either a number (representing a simple weight) or another mobile:

```
function make_branch(length, structure) {
   return list(length, structure);
}
```

a. Write the corresponding selectors `left_branch` and `right_branch`, which return the branches of a mobile, and `branch_length` and `branch_structure`, which return the components of a branch.

b. Using your selectors, define a function `total_weight` that returns the total weight of a mobile.

c. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

d. Suppose we change the representation of mobiles so that the constructors are

```
function make_mobile(left, right) {
   return pair(left, right);
}
function make_branch(length, structure) {
   return pair(length, structure);
}
```

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

Solution

Mapping over trees

Just as `map` is a powerful abstraction for dealing with sequences, `map` together with recursion is a powerful abstraction for dealing with trees. For instance, the `scale_tree` function, analogous to `scale_list` of section 2.1.1, takes as arguments a numeric factor and a tree whose leaves are numbers. It returns a tree of the same shape, where each number is multiplied by the factor. The recursive plan for `scale_tree` is similar to the one for `count_leaves`:

```
function scale_tree(tree, factor) {
    return is_null(tree)
           ? null
           : ! is_pair(tree)
             ? tree * factor
             : pair(scale_tree(head(tree), factor),
                    scale_tree(tail(tree), factor));
}
```

Another way to implement `scale_tree` is to regard the tree as a sequence of sub-trees and use `map`. We map over the sequence, scaling each sub-tree in turn, and return the list of results. In the base case, where the tree is a leaf, we simply multiply by the factor:

```
function scale_tree(tree, factor) {
    return map(sub_tree => is_pair(sub_tree)
                           ? scale_tree(sub_tree, factor)
                           : sub_tree * factor,
               tree);
}
```

Many tree operations can be implemented by similar combinations of sequence operations and recursion.

Exercise 2.14

Define a function `square_tree` analogous to the `square_list` function of exercise 2.5. That is, `square_tree` should behave as follows:

```
square_tree(list(1,
                 list(2, list(3, 4), 5),
                 list(6, 7)));
// result: [1, [[4, [[9, [16, null]],
//                   [25, null]]],
//              [[36, [49, null]], null]]]
```

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

Exercise 2.15

Abstract your answer to exercise 2.14 to produce a function `tree_map` with the property that `square_tree` could be defined as

```
function square_tree(tree) {
   return tree_map(square, tree);
}
```

Solution

Exercise 2.16

We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `[1, [2, [3, null]]]`, then the set of all subsets looks as follows:

```
[null, [[3, null], [[2, null], [[2, [3, null]],
    [[1, null], [[2, [3, null]], [[1, [2, null]],
        [[1, [2, [3, null]]], null]]]]]]]]
```

Complete the following definition of a function that generates the set of subsets of a set and give a clear explanation of why it works:

```
function subsets(s) {
    if (is_null(s)) {
        return list(null);
    } else {
        const rest = subsets(tail(s));
        return append(rest, map(??, rest));
    }
}
```

Solution

2.1.3 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 **??** we saw how program abstractions, implemented as higher-order functions, can capture common patterns in programs that deal with numerical data. Our ability to formulate

analogous operations for working with compound data depends crucially on the style in which we manipulate our data structures. Consider, for example, the following function, analogous to the `count_leaves` function of section 2.1.2, which takes a tree as argument and computes the sum of the squares of the leaves that are odd:

```
function sum_odd_squares(tree) {
    return is_null(tree)
           ? 0
           : ! is_pair(tree)
             ? (is_odd(tree) ? square(tree) : 0)
             : sum_odd_squares(head(tree))
               +
               sum_odd_squares(tail(tree));
}
```

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

```
function even_fibs(n) {
    function next(k) {
        if (k > n) {
            return null;
        } else {
            const f = fib(k);
            return is_even(f)
                   ? pair(f, next(k + 1))
                   : next(k + 1);
        }
    }
    return next(0);
}
```

Despite the fact that these two functions are structurally very different, a more abstract description of the two computations reveals a great deal of similarity. The first program

- enumerates the leaves of a tree;
- filters them, selecting the odd ones;
- squares each of the selected ones; and
- accumulates the results using +, starting with 0.

The second program

- enumerates the integers from 0 to n;

- computes the Fibonacci number for each integer;
- filters them, selecting the even ones; and
- accumulates the results using `pair`, starting with the empty list.

A signal-processing engineer would find it natural to conceptualize these processes in terms of signals flowing through a cascade of stages, each of which implements part of the program plan, as shown in Figure 2.6. In `sum_odd_squares`, we begin with an *enumerator*, which generates a 'signal' consisting of the leaves of a given tree. This signal is passed through a *filter*, which eliminates all but the odd elements. The resulting signal is in turn passed through a *map*, which is a 'transducer' that applies the `square` function to each element. The output of the map is then fed to an *accumulator*, which combines the elements using `+`, starting from an initial `0`. The plan for `even_fibs` is analogous.

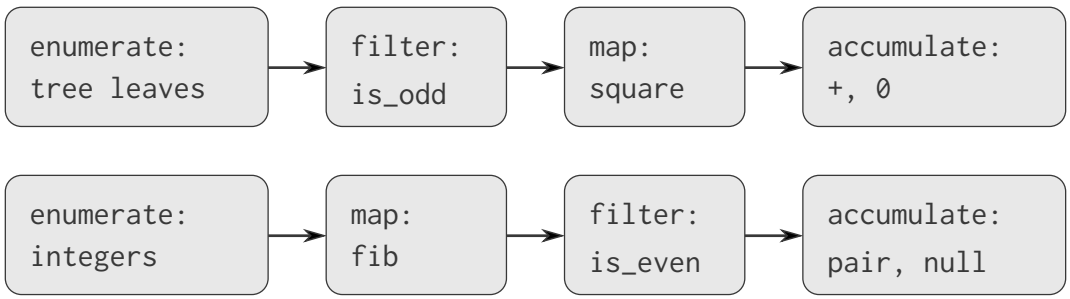


Figure 2.6: The signal-flow plans for the functions `sum_odd_squares` (top) and `even_fibs` (bottom) reveal the commonality between the two programs.

Unfortunately, the two function definitions above fail to exhibit this signal-flow structure. For instance, if we examine the `sum_odd_squares` function, we find that the enumeration is implemented partly by the `is_null` and `is_pair` tests and partly by the tree-recursive structure of the function. Similarly, the accumulation is found partly in the tests and partly in the addition used in the recursion. In general, there are no distinct parts of either function that correspond to the elements in the signal-flow description. Our two functions decompose the computations in a different way, spreading the enumeration over the program and mingling it with the map, the filter, and the accumulation. If we could organize our programs to make the signal-flow structure manifest in the functions we write, this would increase the conceptual clarity of the resulting code.

Sequence Operations

The key to organizing programs so as to more clearly reflect the signal-flow structure is to concentrate on the 'signals' that flow from one stage in the process to the next. If we represent these signals as lists, then we can use list operations to implement the processing at each of the stages. For instance, we can implement the mapping stages of the signal-flow diagrams using the `map` function from section 2.1.1:

```
map(square, list(1, 2, 3, 4, 5));
```

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

```
function filter(predicate, sequence) {
    return is_null(sequence)
           ? null
           : predicate(head(sequence))
             ? pair(head(sequence),
                    filter(predicate, tail(sequence)))
             : filter(predicate, tail(sequence));
}
```

For example,

```
filter(is_odd, list(1, 2, 3, 4, 5));
```

Accumulations can be implemented by

```
function accumulate(op, initial, sequence) {
    return is_null(sequence)
           ? initial
           : op(head(sequence),
                accumulate(op, initial, tail(sequence)));
}
```

```
accumulate(plus, 0, list(1, 2, 3, 4, 5));
```

```
accumulate(times, 1, list(1, 2, 3, 4, 5));
```

```
accumulate(pair, null, list(1, 2, 3, 4, 5));
```

All that remains to implement signal-flow diagrams is to enumerate the sequence of elements to be processed. For `even_fibs`, we need to generate the sequence of integers in a given range, which we can do as follows:

```
function enumerate_interval(low, high) {
    return low > high
           ? null
           : pair(low,
                  enumerate_interval(low + 1, high));
}
```

To enumerate the leaves of a tree, we can use[7]

```
function enumerate_tree(tree) {
    return is_null(tree)
           ? null
           : ! is_pair(tree)
             ? list(tree)
             : append(enumerate_tree(head(tree)),
                      enumerate_tree(tail(tree)));
}
```

Now we can reformulate `sum_odd_squares` and `even_fibs` as in the signal-flow diagrams. For `sum_odd_squares`, we enumerate the sequence of leaves of the tree, filter this to keep only the odd numbers in the sequence, square each element, and sum the results:

```
function sum_odd_squares(tree) {
    return accumulate(plus,
                      0,
                      map(square,
                          filter(is_odd,
                                 enumerate_tree(tree))));
}
```

For `even_fibs`, we enumerate the integers from 0 to n, generate the Fibonacci number for each of these integers, filter the resulting sequence to keep only the even elements, and accumulate the results into a list:

```
function even_fibs(n) {
    return accumulate(pair,
                      null,
                      filter(is_even,
                             map(fib,
                                 enumerate_interval(0, n))));
}
```

The value of expressing programs as sequence operations is that this helps us make program designs that are modular, that is, designs that are constructed by combining relatively independent pieces. We can encourage modular design by providing a library of standard components together with a conventional interface for connecting the components in flexible ways.

[7] This is, in fact, precisely the `fringe` function from exercise 2.12. Here we've renamed it to emphasize that it is part of a family of general sequence-manipulation functions.

Modular construction is a powerful strategy for controlling complexity in engineering design. In real signal-processing applications, for example, designers regularly build systems by cascading elements selected from standardized families of filters and transducers. Similarly, sequence operations provide a library of standard program elements that we can mix and match. For instance, we can reuse pieces from the `sum_odd_squares` and `even-fibs` functions in a program that constructs a list of the squares of the first $n + 1$ Fibonacci numbers:

```
function list_fib_squares(n) {
    return accumulate(pair,
                      null,
                      map(square,
                          map(fib,
                              enumerate_interval(0, n))));
}
```

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

```
function product_of_squares_of_odd_elements(sequence) {
    return accumulate(times,
                      1,
                      map(square,
                          filter(is_odd, sequence)));
}
```

We can also formulate conventional data-processing applications in terms of sequence operations. Suppose we have a sequence of personnel records and we want to find the salary of the highest-paid programmer. Assume that we have a selector `salary` that returns the salary of a record, and a predicate `is_programmer` that tests if a record is for a programmer. Then we can write

```
function salary_of_highest_paid_programmer(records) {
    return accumulate(math_max,
                      0,
                      map(salary,
                          filter(is_programmer, records)));
}
```

These examples give just a hint of the vast range of operations that can be expressed as sequence operations.[8]

Sequences, implemented here as lists, serve as a conventional interface that permits us to com-

[8] Richard Waters (1979) developed a program that automatically analyzes traditional Fortran programs, viewing them in terms of maps, filters, and accumulations. He found that fully 90 percent of the code in the Fortran Scientific Subroutine Package fits neatly into this paradigm. One of the reasons for the success of Lisp as a programming language is that lists provide a standard medium for expressing ordered collections so that they can be manipulated using higher-order operations. The programming language APL owes much of its power and appeal to a similar choice. In APL all data are represented as arrays, and there is a universal and convenient set of generic operators for all sorts of array operations.

bine processing modules. Additionally, when we uniformly represent structures as sequences, we have localized the data-structure dependencies in our programs to a small number of sequence operations. By changing these, we can experiment with alternative representations of sequences, while leaving the overall design of our programs intact. We will exploit this capability in section 3.5, when we generalize the sequence-processing paradigm to admit infinite sequences.

Exercise 2.17

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
function map(f, sequence) {
    return accumulate((x, y) => ?? ,
                      null, sequence);
}

function append(seq1, seq2) {
    return accumulate(pair, ??, ??);
}

function length(sequence) {
    return accumulate(??, 0, sequence);
}
```

Solution

Exercise 2.18

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

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

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

$$(\cdots (a_n x + a_{n-1}) x + \cdots + a_1)\, x + a_0$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0.[9] Fill in the following template to produce a function that evaluates a polynomial

[9] 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

using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
function horner_eval(x, coefficient_sequence) {
    return accumulate((this_coeff, higher_terms) => ??,
                      0,
                      coefficient_sequence);
}
```

For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

```
horner_eval(2, list(1, 3, 0, 5, 0, 1));
```

Solution

Exercise 2.19

Redefine `count_leaves` from section 2.1.2 as an accumulation:

```
function count_leaves(t) {
    return accumulate(??, ??, map(??, ??));
}
```

Solution

Exercise 2.20

The function `accumulate_n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation function to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences

```
list(list( 1,  2,  3),
     list( 4,  5,  6),
     list( 7,  8,  9),
     list(10, 11, 12))
```

then the value of `accumulate_n(plus, 0, s)` should be the sequence `[22, [26, [30, null]]]`. Fill in the missing expressions in the following definition of `accumulate_n`:

must use at least as many additions and multiplications as does Horner's rule, and thus Horner's rule is an optimal algorithm for polynomial evaluation. This was proved (for the number of additions) by A. M. Ostrowski in a 1954 paper that essentially founded the modern study of optimal algorithms. The analogous statement for multiplications was proved by V. Y. Pan in 1966. The book by Borodin and Munro (1975) provides an overview of these and other results about optimal algorithms.

```
function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
           ? null
           : pair(accumulate(op, init, ??),
                  accumulate_n(op, init, ??));
}
```

Solution

Exercise 2.21

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

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

is represented as the following sequence:

```
[[1, [2, [3, [4, null]]]],
 [[4, [5, [6, [6, null]]]],
  [[6, [7, [8, [9, null]]]], null]]]
```

With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

- `dot_product(`v`,` w`)` returns the sum $\sum_i v_i w_i$.
- `matrix_times_vector(`m`,` v`)` returns the vector t, where $t_i = \sum_j m_{ij} v_j$.
- `matrix_times_matrix(`m`,` n`)` returns the matrix p, where $p_{ij} = \sum_k m_{ik} n_{kj}$.
- `transpose(`m`)` returns the matrix n, where $n_{ij} = m_{ji}$.

We can define the dot product as[10]

```
function dot_product(v, w) {
    return accumulate(plus, 0,
                      accumulate_n(times, 1, list(v, w)));
}
```

Fill in the missing expressions in the following functions for computing the other matrix operations. (The function `accumulate_n` is defined in exercise 2.20.)

[10] This definition uses the function `accumulate_n` from exercise 2.20.

```
function matrix_times_vector(m, v) {
    return map(??, m);
}

function transpose(mat) {
    return accumulate_n(??, ??, mat);
}
function matrix_times_matrix(n, m) {
    const cols = transpose(n);
    return map(??, m);
}
```

Solution

Exercise 2.22

The `accumulate` function is also known as `fold_right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold_left`, which is similar to `fold_right`, except that it combines elements working in the opposite direction:

```
function fold_left(op, initial, sequence) {
    function iter(result, rest) {
        return is_null(rest)
               ? result
               : iter(op(result, head(rest)),
                      tail(rest));
    }
    return iter(initial, sequence);
}
```

What are the values of

```
fold_right(divide, 1, list(1, 2, 3));
```

```
fold_left(divide, 1, list(1, 2, 3));
```

```
fold_right(list, null, list(1, 2, 3));
```

```
fold_left(list, null, list(1, 2, 3));
```

Give a property that `op` should satisfy to guarantee that `fold_right` and `fold_left` will produce the same values for any sequence. Solution

Exercise 2.23

Complete the following definitions of `reverse` (exercise 2.2) in terms of `fold_right` and `fold_left` from exercise 2.22:

```
function reverse(sequence) {
    return fold_right((x, y) => ??, null, sequence);
}
```

```
function reverse(sequence) {
    return fold_left((x, y) => ??, null, sequence);
}
```

Solution

Nested Mappings

We can extend the sequence paradigm to include many computations that are commonly expressed using nested loops.[11]

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:[12]

```
accumulate(append,
           null,
           map(i => map(j => list(i, j),
                        enumerate_interval(1, i-1)),
               enumerate_interval(1, n)));
```

[11] 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 2.26) are adapted from Turner 1981. In section 3.5.3, we'll see how this approach generalizes to infinite sequences.

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

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

```
function flatmap(f, seq) {
    return accumulate(append, null, map(f, seq));
}
```

Now filter this sequence of pairs to find those whose sum is prime. The filter predicate is called for each element of the sequence; its argument is a pair and it must extract the integers from the pair. Thus, the predicate to apply to each element in the sequence is

```
function is_prime_sum(pair) {
    return is_prime(head(pair) + head(tail(pair)));
}
```

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

```
function make_pair_sum(pair) {
    return list(head(pair), head(tail(pair)),
                head(pair) + head(tail(pair)));
}
```

Combining all these steps yields the complete function:

```
function prime_sum_pairs(n) {
    return map(make_pair_sum,
              filter(is_prime_sum,
                flatmap(i => map(j => list(i, j),
                                 enumerate_interval(1, i - 1)),
                        enumerate_interval(1, n))));
}
```

Nested mappings are also useful for sequences other than those that enumerate intervals. Suppose we wish to generate all the permutations of a set S; that is, all the ways of ordering the items in the set. For instance, the permutations of $\{1, 2, 3\}$ are $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and $\{3, 2, 1\}$. Here is a plan for generating the permutations of S: For each item x in S, recursively generate the sequence of permutations of $S - x$, [13] 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:[14]

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

[14] The character sequence // in JavaScript code is used to introduce *comments*. Everything from // to the end of the line is ignored by the interpreter. In this book we don't use many comments; we try to make our programs self-documenting by using descriptive names.

```
function permutations(s) {
    return is_null(s)
           ? list(null)
           : flatmap(x => map(p => pair(x, p),
                              permutations(remove(x, s))),
                     s);
}
```

Notice how this strategy reduces the problem of generating permutations of S to the problem of generating the permutations of sets with fewer elements than S. In the terminal case, we work our way down to the empty list, which represents a set of no elements. For this, we generate `list(null)`, which is a sequence with one item, namely the set with no elements. The `remove` function used in `permutations` returns all the items in a given sequence except for a given item. This can be expressed as a simple filter:

```
function remove(item, sequence) {
    return filter(x => !(x === item),
                  sequence);
}
```

Exercise 2.24

Define a function `unique_pairs` that, given an integer n, generates the sequence of pairs (i, j) with $1 \leq j < i \leq n$. Use `unique_pairs` to simplify the definition of `prime_sum_pairs` given above. Solution

Exercise 2.25

Write a function to find all ordered triples of distinct positive integers i, j, and k less than or equal to a given integer n that sum to a given integer s. Solution

Exercise 2.26

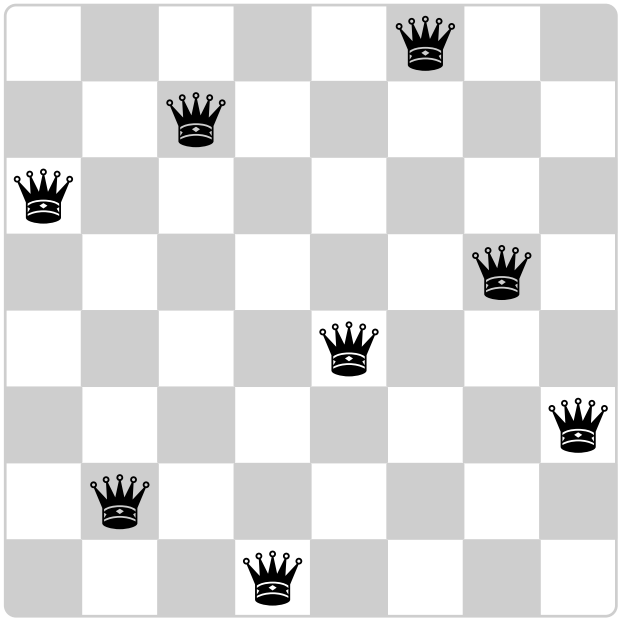


Figure 2.7: A solution to the eight-queens puzzle.

The 'eight-queens puzzle' asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in Figure 2.7. 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 kth 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 kth column. Now filter these, keeping only the positions for which the queen in the kth column is safe with respect to the other queens. This produces the sequence of all ways to place k queens in the first k columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle. We implement this solution as a function queens, which returns a sequence of all solutions to the problem of placing n queens on an $n \times n$ chessboard. The function queens has an internal function queens_cols that returns the sequence of all ways to place queens in the first k columns of the board.

```
function queens(board_size) {
    function queen_cols(k) {
        return k === 0
               ? list(empty_board)
               : filter(
                     positions => is_safe(k, positions),
                     flatmap(rest_of_queens =>
                             map(new_row => adjoin_position(
                                                new_row, k,
                                                rest_of_queens),
```

```
                                enumerate_interval(1,
                                    board_size)),
                            queen_cols(k - 1)));
    }
    return queen_cols(board_size);
}
```

In this function `rest_of_queens` is a way to place $k-1$ queens in the first $k-1$ columns, and `new_row` is a proposed row in which to place the queen for the kth column. Complete the program by implementing the representation for sets of board positions, including the function `adjoin_position`, which adjoins a new row-column position to a set of positions, and `empty_board`, which represents an empty set of positions. You must also write the function `is_safe`, which determines for a set of positions, whether the queen in the kth column is safe with respect to the others. (Note that we need only check whether the new queen is safe—the other queens are already guaranteed safe with respect to each other.) Solution

Exercise 2.27

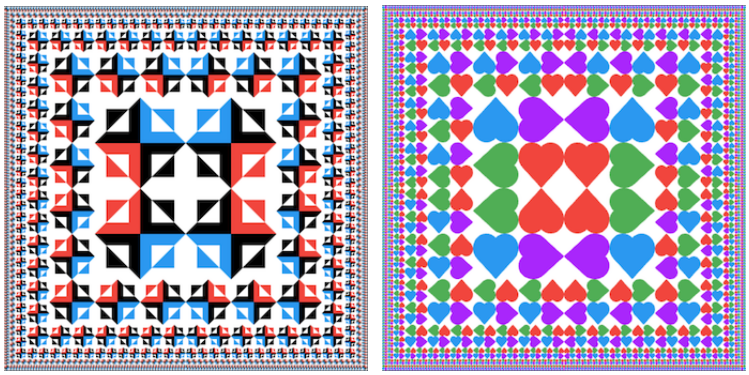
Louis Reasoner is having a terrible time doing exercise 2.26. His `queens` function seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the 6×6 case.) When Louis asks Eva Lu Ator for help, she points out that he has interchanged the order of the nested mappings in the `flatmap`, writing it as

```
flatmap(new_row =>
        map(rest_of_queens => adjoin_position(
                              new_row, k,
                              rest_of_queens),
            queen_cols(k - 1)),
        enumerate_interval(1, board_size));
```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight-queens puzzle, assuming that the program in exercise 2.26 solves the puzzle in time T. Solution

2.1.4 Example: A Picture Language

This section presents a simple language for drawing pictures that illustrates the power of data abstraction and closure, and also exploits higher-order functions in an essential way. The language is designed to make it easy to experiment with patterns such as the ones in figure 2.8, which are composed of repeated elements that are shifted and scaled.[15] In this language, the data objects being combined are represented as functions rather than as list structure. Just as `pair`, which satisfies the closure property, allowed us to easily build arbitrarily complicated list structure, the operations in this language, which also satisfy the closure property, allow us to easily build arbitrarily complicated patterns.



(a) (b)

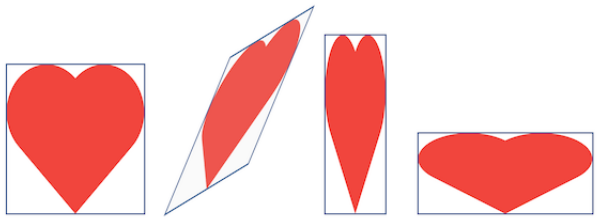
Figure 2.8: Designs generated with the picture language.

The picture language

When we began our study of programming in section **??**, we emphasized the importance of describing a language by focusing on the language's primitives, its means of combination, and its means of abstraction. We'll follow that framework here.

Part of the elegance of this picture language is that there is only one kind of element, called a *painter*. A painter draws an image that is shifted and scaled to fit within a designated parallelogram-shaped frame. For example, there's a primitive painter we'll call `heart` that makes a heart shape, as shown in figure 2.9.

[15] 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` function in this section.



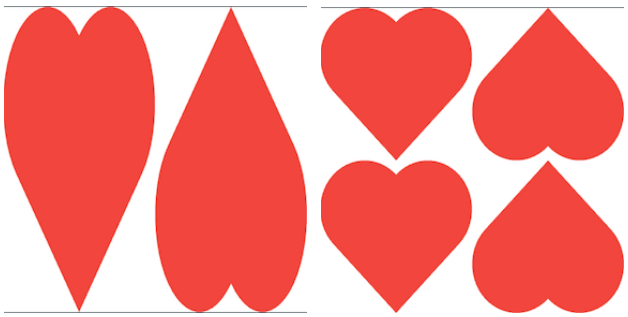
(a)

Figure 2.9: Images produced by the `heart` painter, with respect to four different frames. The frames, shown with thin lines, are not part of the images.

The actual shape of the drawing depends on the frame—all four images in figure 2.9 are produced by the same `heart` painter, but with respect to four different frames. In the following, we shall use a function `show` to display a painter in a default frame.

```
show(heart);
```

To combine images, we use various operations that construct new painters from given painters. For example, the `beside` operation takes two painters and produces a new, compound painter that draws the first painter's image in the left half of the frame and the second painter's image in the right half of the frame. Similarly, `stack` takes two painters and produces a compound painter that draws the first painter's image below the second painter's image. Some operations transform a single painter to produce a new painter. For example, `flip_vert` takes a painter and produces a painter that draws its image upside-down, and `flip_horiz` produces a painter that draws the original painter's image left-to-right reversed.



(a) (b)

Figure 2.10: Creating a complex figure, starting from the `heart` painter of Figure 2.9.

Figure 2.10 shows the drawing of a painter called `heart4` that is built up in two stages starting from `heart`:

```
const heart2 = beside(heart, flip_vert(heart)); // (a)
```

```
const heart4 = stack(heart2, heart2);           // (b)
```

In building up a complex image in this manner we are exploiting the fact that painters are

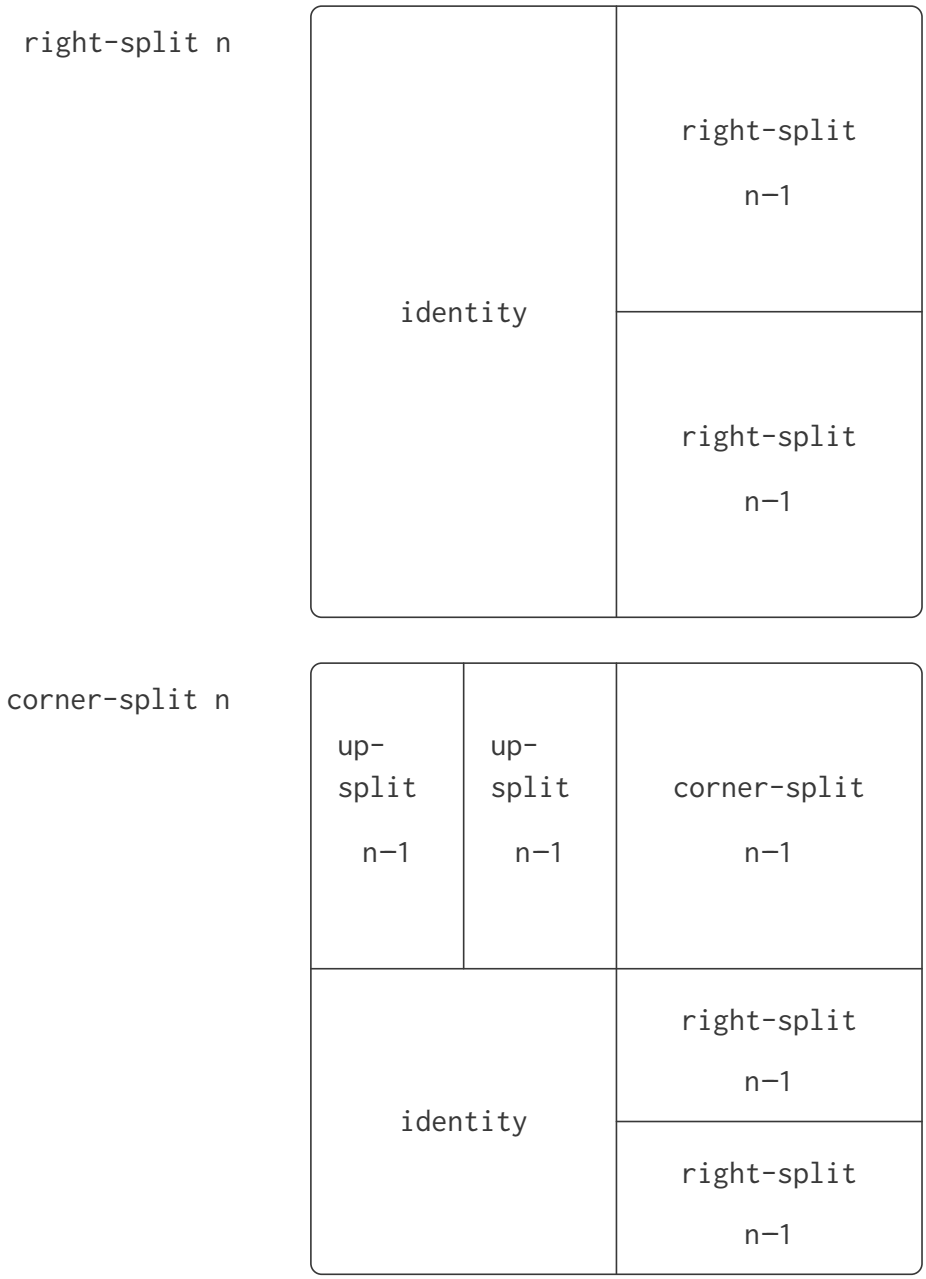
closed under the language's means of combination. The `beside` or `stack` of two painters is itself a painter; therefore, we can use it as an element in making more complex painters. As with building up list structure using `pair`, the closure of our data under the means of combination is crucial to the ability to create complex structures while using only a few operations.

Once we can combine painters, we would like to be able to abstract typical patterns of combining painters. We will implement the painter operations as JavaScript functions. This means that we don't need a special abstraction mechanism in the picture language: Since the means of combination are ordinary JavaScript functions, we automatically have the capability to do anything with painter operations that we can do with functions. For example, we can abstract the pattern in `wave4` as

```
function flipped_pairs(painter) {
    const painter2 = beside(painter, flip_vert(painter));
    return stack(painter2, painter2);
}
```

and define `heart4` as an instance of this pattern:

```
const heart4 = flipped_pairs(heart);
```



(a)

Figure 2.11: Recursive plans for (a) `right_split(n)` and (b) `corner_split(n)`.

We can also define recursive operations. Here's one that makes painters split and branch towards the right as shown in figures 2.11, 2.12 and 2.13:

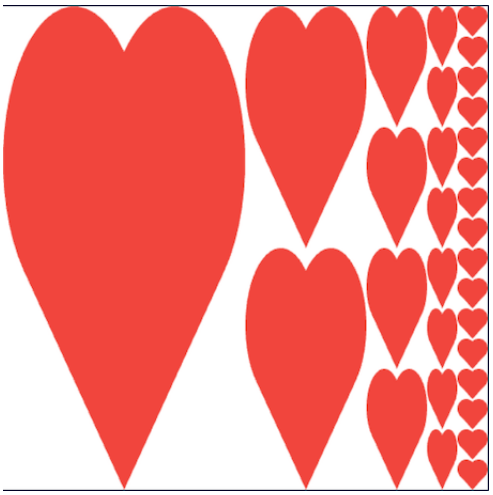
```
function right_split(painter, n) {
    if (n === 0) {
        return painter;
    } else {
        const smaller = right_split(painter, n - 1);
        return beside(painter, stack(smaller, smaller));
    }
```

```
}
```

We can produce balanced patterns by branching upwards as well as towards the right (see exercise 2.28 and Figure 2.11).

```
function corner_split(painter, n) {
    if (n === 0) {
       return painter;
    } else {
       const up = up_split(painter, n - 1);
       const right = right_split(painter, n - 1);
       const top_left = beside(up, up);
       const bottom_right = stack(right, right);
       const corner = corner_split(painter, n - 1);
       return stack(beside(top_left, corner),
                    beside(painter, bottom_right));
    }
}
```

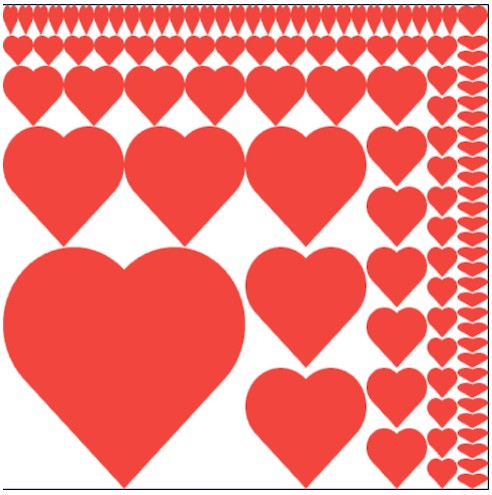
```
show(right_split(heart, 4));
```



(a)

Figure 2.12: The recursive operation `right_split` applied to the painter `heart`.

```
show(corner_split(heart, 4));
```



(a)

Figure 2.13: The recursive operation `corner_split` applied to the painter `heart`. Combining four `corner_split` figures produces symmetric `square_limit` as shown in figure 2.8.

By placing four copies of a `corner_split` appropriately, we obtain a pattern called `square_limit`, whose application to two painters is shown in figure 2.8:

```
function square_limit(painter, n) {
    const quarter = corner_split(painter, n);
    const upper_half = beside(flip_horiz(quarter), quarter);
    const lower_half = beside(turn_upside_down(quarter),
                              flip_vert(quarter));
    return stack(upper_half, lower_half);
}
```

Exercise 2.28

Define the function `up_split` used by `corner_split`. It is similar to `right_split`, except that it switches the roles of `stack` and `beside`. Solution

Higher-order operations

In addition to abstracting patterns of combining painters, we can work at a higher level, abstracting patterns of combining painter operations. That is, we can view the painter operations as elements to manipulate and can write means of combination for these elements—functions that take painter operations as arguments and create new painter operations.

For example, `flipped_pairs` and `square_limit` each arrange four copies of a painter's image in a square pattern; they differ only in how they orient the copies. One way to abstract this pattern of painter combination is with the following function, which takes four one-argument

painter operations and produces a painter operation that transforms a given painter with those four operations and arranges the results in a square. The functions `tl`, `tr`, `bl`, and `br` are the transformations to apply to the top left copy, the top right copy, the bottom left copy, and the bottom right copy, respectively.

```
function square_of_four(tl, tr, bl, br) {
    return painter => stack(beside(tl(painter), tr(painter)),
                            beside(bl(painter), br(painter)));
}
```

Then `flipped_pairs` can be defined in terms of `square_of_four` as follows:[16]

```
function flipped_pairs(painter) {
    const combine4 = square_of_four(turn_upside_down, flip_vert,
                                    flip_horiz, identity);
    return combine4(painter);
}
```

and `square_limit` can be expressed as[17]

```
function square_limit(painter, n) {
    const combine4 = square_of_four(flip_horiz, identity,
                                    turn_upside_down, flip_vert);
    return combine4(corner_split(painter, n));
}
```

Exercise 2.29

The functions `right_split` and `up_split` can be expressed as instances of a general splitting operation. Define a function `split` with the property that evaluating

```
const right_split = split(beside, below);
const up_split = split(below, beside);
```

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

[16] Equivalently, we could write

```
const flipped_pairs =
    square_of_four(turn_upside_down, flip_vert,
                   flip_horiz, identity);
```

[17] The function `turn_upside_down` rotates a painter by 180 degrees. Instead of `turn_upside_down` we could say `compose(flip_vert, flip_horiz)`, using the `compose` function from exercise ??.

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 2.14 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 2.31).

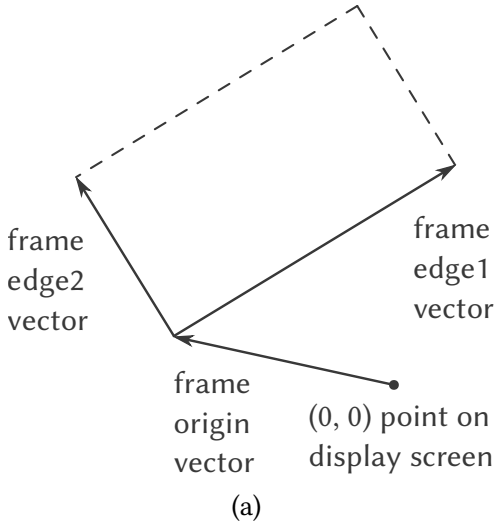


Figure 2.14: A frame is described by three vectors—an origin and two edges.

We will use coordinates in the unit square ($0 \leq x, y \leq 1$) to specify images. With each frame, we associate a *frame coordinate map*, which will be used to shift and scale images to fit the frame. The map transforms the unit square into the frame by mapping the vector $\mathbf{v} = (x, y)$ to the vector sum

$$\text{Origin(Frame)} + x \cdot \text{Edge}_1\text{(Frame)} + y \cdot \text{Edge}_2\text{(Frame)}$$

For example, $(0, 0)$ is mapped to the origin of the frame, $(1, 1)$ to the vertex diagonally opposite the origin, and $(0.5, 0.5)$ to the center of the frame. We can create a frame's coordinate map with the following function:[18]

[18]The function `frame_coord_map` uses the vector operations described in exercise 2.30 below, which we assume have been implemented using some representation for vectors. Because of data abstraction, it doesn't matter what this vector representation is, so long as the vector operations behave correctly.

```
function frame_coord_map(frame) {
    return v => add_vect(origin_frame(frame),
                         add_vect(scale_vect(xcor_vect(v),
                                             edge1_frame(frame)),
                                  scale_vect(ycor_vect(v),
                                             edge2_frame(frame))));
}
```

Observe that applying `frame_coord_map` to a frame returns a function that, given a vector, returns a vector. If the argument vector is in the unit square, the result vector will be in the frame. For example,

```
frame_coord_map(a_frame)(make_vect(0, 0));
```

returns the same vector as

```
origin_frame(a_frame);
```

Exercise 2.30

A two-dimensional vector v running from the origin to a point can be represented as a pair consisting of an x-coordinate and a y-coordinate. Implement a data abstraction for vectors by giving a constructor `make_vect` and corresponding selectors `xcor_vect` and `ycor_vect`. In terms of your selectors and constructor, implement functions `add_vect`, `sub_vect`, and `scale_vect` that perform the operations vector addition, vector subtraction, and multiplying a vector by a scalar:

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

Solution

Exercise 2.31

Here are two possible constructors for frames:

```
function make_frame(origin, edge1, edge2) {
    return list(origin, edge1, edge2);
}

function make_frame(origin, edge1, edge2) {
    return pair(origin, pair(edge1, edge2));
}
```

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

Painters

A painter is represented as a function that, given a frame as argument, draws a particular image shifted and scaled to fit the frame. That is to say, if `p` is a painter and `f` is a frame, then we produce `p`'s image in `f` by calling `p` with `f` as argument.

The details of how primitive painters are implemented depend on the particular characteristics of the graphics system and the type of image to be drawn. For instance, suppose we have a function `draw_line` that draws a line on the screen between two specified points. Then we can create painters for line drawings, such as the `wave` painter in figure 2.9, from lists of line segments as follows:[19]

```
function segments_to_painter(segment_list) {
    return frame =>
               for_each(segment =>
                            draw_line(frame_coord_map(frame)
                                      (start_segment(segment)),
                                      frame_coord_map(frame)
                                      (end_segment(segment))),
                        segment_list);
}
```

The segments are given using coordinates with respect to the unit square. For each segment in the list, the painter transforms the segment endpoints with the frame coordinate map and draws a line between the transformed points.

Representing painters as functions erects a powerful abstraction barrier in the picture language. We can create and intermix all sorts of primitive painters, based on a variety of graphics capabilities. The details of their implementation do not matter. Any function can serve as a painter, provided that it takes a frame as argument and draws something scaled to fit the frame.[20]

[19] The function `segments_to_painter` uses the representation for line segments described in exercise 2.32 below. It also uses the `for_each` function described in exercise 2.7.

[20] For example, the `heart` painter of figure 2.9 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 **??**, 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 **??** also showed how pairs could be implemented as functions. Painters are our second example of a functional representation for data.

Exercise 2.32

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 2.30 to define a representation for segments with a constructor `make_segment` and selectors `start_segment` and `end_segment`. Solution

Exercise 2.33

Use `segments_to_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.

Solution

Transforming and combining painters

An operation on painters (such as `flip_vert` or `beside`) works by creating a painter that invokes the original painters with respect to frames derived from the argument frame. Thus, for example, `flip_vert` doesn't have to know how a painter works in order to flip it—it just has to know how to turn a frame upside down: The flipped painter just uses the original painter, but in the inverted frame.

Painter operations are based on the function `transform_painter`, which takes as arguments a painter and information on how to transform a frame and produces a new painter. The transformed painter, when called on a frame, transforms the frame and calls the original painter on the transformed frame. The arguments to `transform_painter` are points (represented as vectors) that specify the corners of the new frame: When mapped into the frame,[21] the first

[21] In `transform_painter`, we make use of a slight extension of the syntax of function definition expressions, compared to section **??**: The body of a function definition can be a block, not just a single return expression. Such function definition expressions have the following shape:

(*parameters*) => { *statements* }

point specifies the new frame's origin and the other two specify the ends of its edge vectors. Thus, arguments within the unit square specify a frame contained within the original frame.

```
function transform_painter(painter, origin,
                           corner1, corner2) {
    return frame => {
             const m = frame_coord_map(frame);
             const new_origin = m(origin);
             return painter(make_frame(
                              new_origin,
                              sub_vect(m(corner1),
                                       new_origin),
                              sub_vect(m(corner2),
                                       new_origin)));
           };
}
```

Here's how to flip painter images vertically:

```
function flip_vert(painter) {
    return transform_painter(painter,
                make_vect(0.0, 1.0),  // new origin
                make_vect(1.0, 1.0),  // new end of edge1
                make_vect(0.0, 0.0)); // new end of edge2
}
```

Using `transform_painter`, we can easily define new transformations. For example, we can define a painter that shrinks its image to the upper-right quarter of the frame it is given:

```
function shrink_to_upper_right(painter) {
    return transform_painter(painter,
                             make_vect(0.5, 0.5),
                             make_vect(1.0, 0.5),
                             make_vect(0.5, 1.0));
}
```

Other transformations rotate images counterclockwise by 90 degrees[22]

```
function rotate90(painter) {
    return transform_painter(painter,
                             make_vect(1.0, 0.0),
                             make_vect(1.0, 1.0),
                             make_vect(0.0, 0.0));
}
```

or squash images towards the center of the frame:[23]

[22] The function `rotate90` is a pure rotation only for square frames, because it also stretches and shrinks the image to fit into the rotated frame.

[23] The diamond-shaped images in figures 2.9 were created with `squash_inwards` applied to `heart`.

```
function squash_inwards(painter) {
    return transform_painter(painter,
                             make_vect(0.0, 0.0),
                             make_vect(0.65, 0.35),
                             make_vect(0.35, 0.65));
}
```

Frame transformation is also the key to defining means of combining two or more painters. The `beside` function, for example, takes two painters, transforms them to paint in the left and right halves of an argument frame respectively, and produces a new, compound painter. When the compound painter is given a frame, it calls the first transformed painter to paint in the left half of the frame and calls the second transformed painter to paint in the right half of the frame:

```
function beside(painter1, painter2) {
    const split_point = make_vect(0.5, 0.0);
    const paint_left = transform_painter(painter1,
                            make_vect(0.0, 0.0),
                            split_point,
                            make_vect(0.0, 1.0));
    const paint_right = transform_painter(painter2,
                            split_point,
                            make_vect(1.0, 0.0),
                            make_vect(0.5, 1.0));
    return frame => {
                paint_left(frame);
                paint_right(frame);
            };
}
```

Observe how the painter data abstraction, and in particular the representation of painters as functions, makes `beside` easy to implement. The `beside` function need not know anything about the details of the component painters other than that each painter will draw something in its designated frame.

Exercise 2.34

Define the transformation `flip_horiz`, which flips painters horizontally, and transformations that rotate painters counterclockwise by 180 degrees and 270 degrees. Solution

Exercise 2.35

Define the `stack` operation for painters. The function `stack` takes two painters as arguments. The resulting painter, given a frame, draws with the first painter in the bottom of the frame

and with the second painter in the top. Define `stack` in two different ways—first by writing a function that is analogous to the `beside` function given above, and again in terms of `beside` and suitable rotation operations (from exercise 2.34). Solution

Levels of language for robust design

The picture language exercises some of the critical ideas we've introduced about abstraction with functions and data. The fundamental data abstractions, painters, are implemented using functional representations, which enables the language to handle different basic drawing capabilities in a uniform way. The means of combination satisfy the closure property, which permits us to easily build up complex designs. Finally, all the tools for abstracting functions are available to us for abstracting means of combination for painters.

We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of *stratified design*, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail.

Stratified design pervades the engineering of complex systems. For example, in computer engineering, resistors and transistors are combined (and described using a language of analog circuits) to produce parts such as and-gates and or-gates, which form the primitives of a language for digital-circuit design.[24] These parts are combined to build processors, bus structures, and memory systems, which are in turn combined to form computers, using languages appropriate to computer architecture. Computers are combined to form distributed systems, using languages appropriate for describing network interconnections, and so on.

As a tiny example of stratification, our picture language uses primitive elements (primitive painters) that specify points and lines to provide the shapes of a painter like `heart`. The bulk of our description of the picture language focused on combining these primitives, using geometric combiners such as `beside` and `stack`. We also worked at a higher level, regarding `beside` and `stack` as primitives to be manipulated in a language whose operations, such as `square_of_four`, capture common patterns of combining geometric combiners.

Stratified design helps make programs *robust*, that is, it makes it likely that small changes in a specification will require correspondingly small changes in the program. For instance, suppose we wanted to change the image based on `heart` shown in Figure 2.8. We could work at the

[24] Section 3.3.4 describes one such language.

lowest level to change the detailed appearance of the `heart` element; we could work at the middle level to change the way `corner_split` replicates the `wave`; we could work at the highest level to change how `square_limit` arranges the four copies of the corner. In general, each level of a stratified design provides a different vocabulary for expressing the characteristics of the system, and a different kind of ability to change it.

Exercise 2.36

Make changes to the square limit of `heart` shown in Figure 2.8 by working at each of the levels described above. In particular:

a. Change the pattern constructed by `corner_split` (for example, by using only one copy of the `up_split` and `right_split` images instead of two).

b. Modify the version of `square_limit` that uses `square_of_four` so as to assemble the corners in a different pattern.

Solution

2.2 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 **??** how to separate the task of designing a program that uses rational numbers from the task of implementing rational numbers in terms of the computer language's primitive mechanisms for constructing compound data. The key idea was to erect an abstraction barrier—in this case, the selectors and constructors for rational numbers (`make_rat`, `numer`, `denom`)—that isolates the way rational numbers are used from their underlying representation in terms of list structure. A similar abstraction barrier isolates the details of the functions that perform rational arithmetic (`add_rat`, `sub_rat`, `mul_rat`, and `div_rat`) from the 'higher-level' functions that use rational numbers. The resulting program has the structure shown in figure **??**.

These data-abstraction barriers are powerful tools for controlling complexity. By isolating the underlying representations of data objects, we can divide the task of designing a large program into smaller tasks that can be performed separately. But this kind of data abstraction is not yet powerful enough, because it may not always make sense to speak of 'the underlying representation' for a data object.

For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations. To take a simple example, complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle). Sometimes rectangular form is more appropriate and sometimes polar form is more appropriate. Indeed, it is perfectly plausible to imagine a system in which complex numbers are represented in both ways, and in which the functions for manipulating complex numbers work with either representation.

More importantly, programming systems are often designed by many people working over extended periods of time, subject to requirements that change over time. In such an environment, it is simply not possible for everyone to agree in advance on choices of data representation. So in addition to the data-abstraction barriers that isolate representation from use, we need abstraction barriers that isolate different design choices from each other and permit different choices to coexist in a single program. Furthermore, since large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems *additively*, that is, without having to redesign or reimplement these modules.

In this section, we will learn how to cope with data that may be represented in different ways by different parts of a program. This requires constructing *generic functions*—functions that can operate on data that may be represented in more than one way. Our main technique for building generic functions will be to work in terms of data objects that have *type tags*, that is, data objects that include explicit information about how they are to be processed. We will also discuss *data-directed* programming, a powerful and convenient implementation strategy for additively assembling systems with generic operations.

We begin with the simple complex-number example. We will see how type tags and data-directed style enable us to design separate rectangular and polar representations for complex numbers while maintaining the notion of an abstract 'complex-number' data object. We will accomplish this by defining arithmetic functions for complex numbers (`add_complex`, `sub_complex`, `mul_complex`, and `div_complex`) in terms of generic selectors that access parts of a complex number independent of how the number is represented. The resulting complex-number system, as shown in figure 2.15, contains two different kinds of abstraction barriers. The 'horizontal' abstraction barriers play the same role as the ones in figure **??**. They isolate 'higher-level' operations from 'lower-level' representations. In addition, there is a 'vertical' barrier that gives us the ability to separately design and install alternative representations.

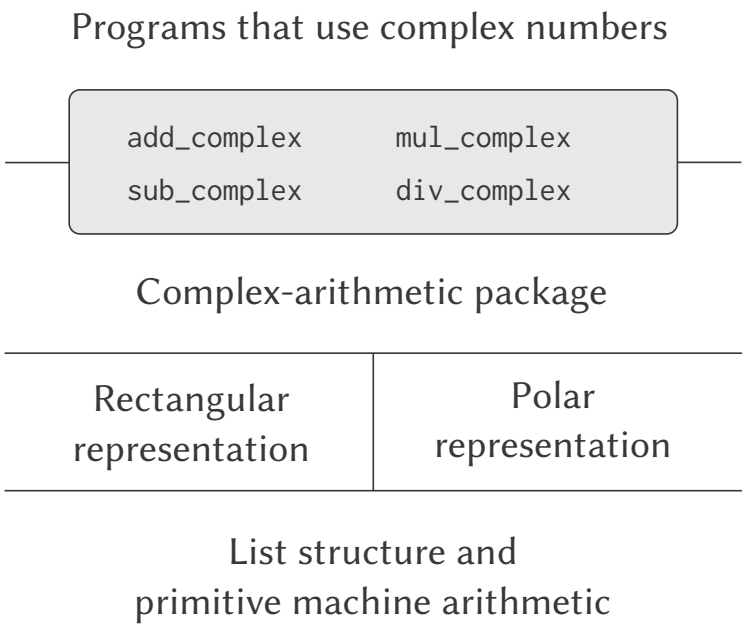


Figure 2.15: Data-abstraction barriers in the complex-number system.

In section **??** we will show how to use type tags and data-directed style to develop a generic arithmetic package. This provides functions (`add`, `mul`, and so on) that can be used to manipulate all sorts of 'numbers' and can be easily extended when a new kind of number is needed. In section **??**, we'll show how to use generic arithmetic in a system that performs symbolic algebra.

2.2.1 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).[25] Section 2.2.2 will show how both representations can be made to coexist in a single system through the use of type tags and generic operations.

Like rational numbers, complex numbers are naturally represented as ordered pairs. The set of complex numbers can be thought of as a two-dimensional space with two orthogonal axes, the 'real' axis and the 'imaginary' axis. (See figure 2.16.) 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

[25] 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.

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 2.16). 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.

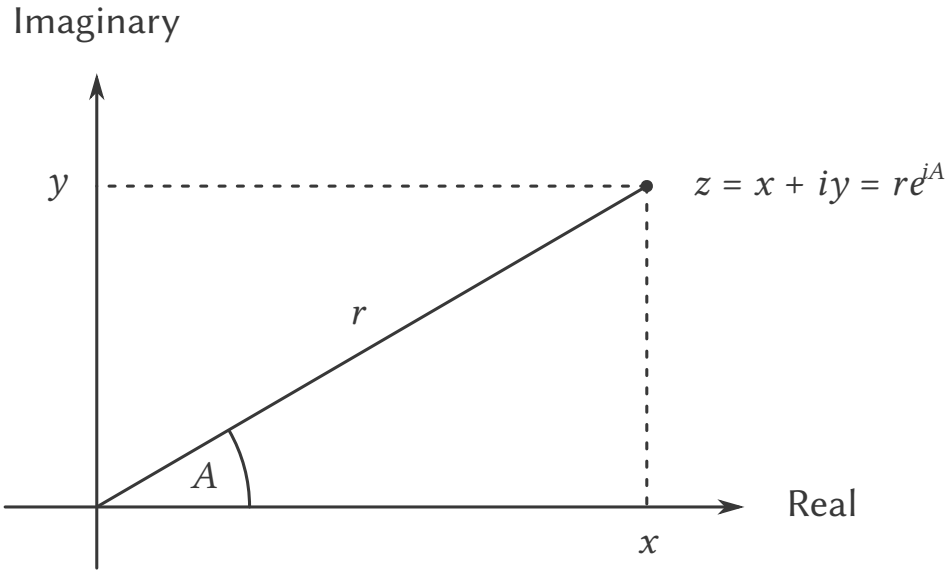


Figure 2.16: Complex numbers as points in the plane.

To design such a system, we can follow the same data-abstraction strategy we followed in designing the rational-number package in section **??**. Assume that the operations on complex numbers are implemented in terms of four selectors: `real_part`, `imag_part`, `magnitude`, and

angle. Also assume that we have two functions for constructing complex numbers: make_from_real_imag returns a complex number with specified real and imaginary parts, and make_from_mag_ang returns a complex number with specified magnitude and angle. These functions have the property that, for any complex number z, both

```
make_from_real_imag(real_part(z),imag_part(z));
```

and

```
make_from_mag_ang(magnitude(z), angle(z));
```

produce complex numbers that are equal to z.

Using these constructors and selectors, we can implement arithmetic on complex numbers using the 'abstract data' specified by the constructors and selectors, just as we did for rational numbers in section **??**. As shown in the formulas above, we can add and subtract complex numbers in terms of real and imaginary parts while multiplying and dividing complex numbers in terms of magnitudes and angles:

```
function add_complex(z1, z2) {
    return make_from_real_imag(
               real_part(z1) + real_part(z2),
               imag_part(z1) + imag_part(z2));
}
function sub_complex(z1, z2) {
    return make_from_real_imag(
               real_part(z1) - real_part(z2),
               imag_part(z1) - imag_part(z2));
}
function mul_complex(z1, z2) {
    return make_from_mag_ang(
               magnitude(z1) * magnitude(z2),
               angle(z1) + angle(z2));
}
function div_complex(z1, z2) {
    return make_from_mag_ang(
               magnitude(z1) / magnitude(z2),
               angle(z1) - angle(z2));
}
```

To complete the complex-number package, we must choose a representation and we must implement the constructors and selectors in terms of primitive numbers and primitive list structure. There are two obvious ways to do this: We can represent a complex number in 'rectangular form' as a pair (real part, imaginary part) or in 'polar form' as a pair (magnitude, angle). Which shall we choose?

In order to make the different choices concrete, imagine that there are two programmers, Ben Bitdiddle and Alyssa P. Hacker, who are independently designing representations for the

complex-number system. Ben chooses to represent complex numbers in rectangular form. With this choice, selecting the real and imaginary parts of a complex number is straightforward, as is constructing a complex number with given real and imaginary parts. To find the magnitude and the angle, or to construct a complex number with a given magnitude and angle, he uses the trigonometric relations

$$x = r\ \cos A \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\ \sin A \qquad A = \arctan(y, x)$$

which relate the real and imaginary parts (x, y) to the magnitude and the angle (r, A).[26] Ben's representation is therefore given by the following selectors and constructors:

```
function real_part(z) {
    return head(z);
}
function imag_part(z) {
    return tail(z);
}
function magnitude(z) {
    return math_sqrt(
               square(real_part(z)) +
               square(imag_part(z)));
}
function angle(z) {
    return math_atan2(imag_part(z),real_part(z));
}
function make_from_real_imag(x, y) {
    return pair(x, y);
}
function make_from_mag_ang(r, a) {
    return pair(r * math_cos(a), r * math_sin(a));
}
```

Alyssa, in contrast, chooses to represent complex numbers in polar form. For her, selecting the magnitude and angle is straightforward, but she has to use the trigonometric relations to obtain the real and imaginary parts. Alyssa's representation is:

```
function real_part(z) {
    return magnitude(z) * math_cos(angle(z));
}
function imag_part(z) {
    return magnitude(z) * math_sin(angle(z));
}
```

[26] The arctangent function referred to here, computed by JavaScript's `math_atan2` function, is defined so as to take two arguments y and x and to return the angle whose tangent is y/x. The signs of the arguments determine the quadrant of the angle.

```
function magnitude(z) {
    return head(z);
}
function angle(z) {
    return tail(z);
}
function make_from_real_imag(x, y) {
    return pair(math_sqrt(square(x) + square(y)),
                math_atan2(y, x));
}
function make_from_mag_ang(r, a) {
    return pair(r, a);
}
```

The discipline of data abstraction ensures that the same implementation of `add_complex`, `sub_complex`, `mul_complex`, and `div_complex` will work with either Ben's representation or Alyssa's representation.

2.2.2 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 2.2.1, we can use either Ben's rectangular representation or Alyssa's polar representation. The abstraction barrier formed by the selectors and constructors permits us to defer to the last possible moment the choice of a concrete representation for our data objects and thus retain maximum flexibility in our system design.

The principle of least commitment can be carried to even further extremes. If we desire, we can maintain the ambiguity of representation even *after* we have designed the selectors and constructors, and elect to use both Ben's representation *and* Alyssa's representation. If both representations are included in a single system, however, we will need some way to distinguish data in polar form from data in rectangular form. Otherwise, if we were asked, for instance, to find the `magnitude` of the pair $(3, 4)$, we wouldn't know whether to answer 5 (interpreting the number in rectangular form) or 3 (interpreting the number in polar form). A straightforward way to accomplish this distinction is to include a *type tag*—the symbol `rectangular` or `polar`—as part of each complex number. Then when we need to manipulate a complex number we can use the tag to decide which selector to apply.

In order to manipulate tagged data, we will assume that we have functions `type_tag` and `contents` that extract from a data object the tag and the actual contents (the polar or rectangular coordinates, in the case of a complex number). We will also postulate a function `attach_tag` that takes a tag and contents and produces a tagged data object. A straightforward way to implement this is to use ordinary list structure:

```
function attach_tag(type_tag, contents) {
    return pair(type_tag, contents);
}
function type_tag(datum) {
    return is_pair(datum)
           ? head(datum)
           : Error("bad tagged datum in type_tag", datum);
}
function contents(datum) {
    return is_pair(datum)
           ? tail(datum)
           : Error("bad tagged datum in contents", datum);
}
```

Using these functions, we can define predicates `is_rectangular` and `is_polar`, which recognize polar and rectangular numbers, respectively:

```
function is_rectangular(z) {
    return type_tag(z) === "rectangular";
}
function is_polar(z) {
    return type_tag(z) === "polar";
}
```

With type tags, Ben and Alyssa can now modify their code so that their two different representations can coexist in the same system. Whenever Ben constructs a complex number, he tags it as rectangular. Whenever Alyssa constructs a complex number, she tags it as polar. In addition, Ben and Alyssa must make sure that the names of their functions do not conflict. One way to do this is for Ben to append the suffix `rectangular` to the name of each of his representation functions and for Alyssa to append `polar` to the names of hers. Here is Ben's revised rectangular representation from section 2.2.1:

```
function real_part_rectangular(z) {
    return head(z);
}
function imag_part_rectangular(z) {
    return tail(z);
}
function magnitude_rectangular(z) {
    return math_sqrt(square(real_part_rectangular(z))
           +
           square(imag_part_rectangular(z)));
}
function angle_rectangular(z) {
    return math_atan(imag_part_rectangular(z),
                real_part_rectangular(z));
}
```

```
function make_from_real_imag_rectangular(x, y) {
    return attach_tag("rectangular",
                      pair(x, y));
}
function make_from_mag_ang_rectangular(r, a) {
    return attach_tag("rectangular",
                      pair(r * math_cos(a), r * math_sin(a)));
}
```

and here is Alyssa's revised polar representation:

```
function real_part_polar(z) {
    return magnitude_polar(z) * math_cos(angle_polar(z));
}
function imag_part_polar(z) {
    return magnitude_polar(z) * math_sin(angle_polar(z));
}
function magnitude_polar(z) {
    return head(z);
}
function angle_polar(z) {
    return tail(z);
}
function make_from_real_imag_polar(x, y) {
    return attach_tag("polar",
                      pair(math_sqrt(square(x) + square(y)),
                           math_atan(y, z)));
}
function make_from_mag_ang_polar(r, a) {
    return attach_tag("polar",
                      pair(r, a));
}
```

Each generic selector is implemented as a function that checks the tag of its argument and calls the appropriate function for handling data of that type. For example, to obtain the real part of a complex number, `real_part` examines the tag to determine whether to use Ben's `real_part_rectangular` or Alyssa's `real_part_polar`. In either case, we use `contents` to extract the bare, untagged datum and send this to the rectangular or polar function as required:

```
function real_part(z) {
    return is_rectangular(z)
           ? real_part_rectangular(contents(z))
           : is_polar(z)
             ? real_part_polar(contents(z))
             : Error("Unknown type in real_part", z);
}
function imag_part(z) {
    return is_rectangular(z)
```

```
           ? imag_part_rectangular(contents(z))
           : is_polar(z)
             ? imag_part_polar(contents(z))
             : Error("Unknown type in imag_part", z);
}
function magnitude(z) {
    return is_rectangular(z)
           ? magnitude_rectangular(contents(z))
           : is_polar(z)
             ? magnitude_polar(contents(z))
             : Error("Unknown type in magnitude", z);
}
function angle(z) {
    return is_rectangular(z)
           ? angle_rectangular(contents(z))
           : is_polar(z)
             ? angle_polar(contents(z))
             : Error("Unknown type in angle", z);
}
```

To implement the complex-number arithmetic operations, we can use the same functions `add_complex`, `sub_complex`, `mul_complex`, and `div_complex` from section 2.2.1, because the selectors they call are generic, and so will work with either representation. For example, the function `add_complex` is still

```
function add_complex(z1, z2) {
   return make_from_real_imag(
             real_part(z1) + real_part(z2),
             imag_part(z1) + imag_part(z2));
}
```

Finally, we must choose whether to construct complex numbers using Ben's representation or Alyssa's representation. One reasonable choice is to construct rectangular numbers whenever we have real and imaginary parts and to construct polar numbers whenever we have magnitudes and angles:

```
function make_from_real_imag(x, y) {
    return make_from_real_imag_rectangular(x, y);
}
function make_from_mag_ang(r, a) {
    return make_from_mag_ang_polar(r, a);
}
```

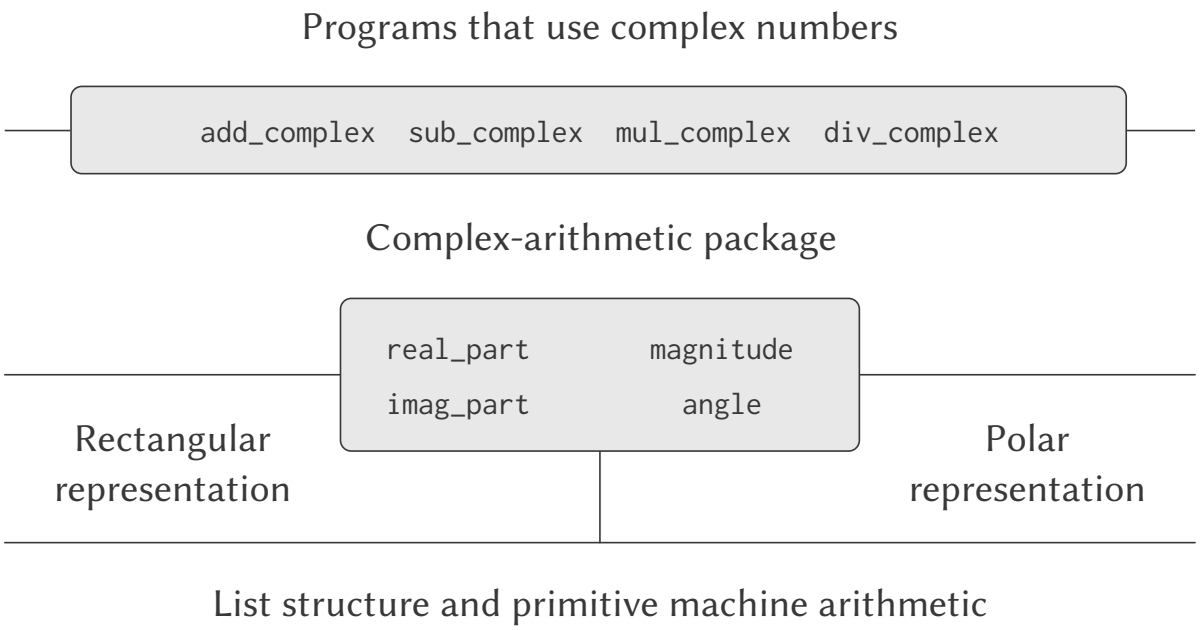


Figure 2.17: Structure of the generic complex-arithmetic system.

The resulting complex-number system has the structure shown in figure 2.17. The system has been decomposed into three relatively independent parts: the complex-number-arithmetic operations, Alyssa's polar implementation, and Ben's rectangular implementation. The polar and rectangular implementations could have been written by Ben and Alyssa working separately, and both of these can be used as underlying representations by a third programmer implementing the complex-arithmetic functions in terms of the abstract constructor/selector interface.

Since each data object is tagged with its type, the selectors operate on the data in a generic manner. That is, each selector is defined to have a behavior that depends upon the particular type of data it is applied to. Notice the general mechanism for interfacing the separate representations: Within a given representation implementation (say, Alyssa's polar package) a complex number is an untyped pair (magnitude, angle). When a generic selector operates on a number of `polar` type, it strips off the tag and passes the contents on to Alyssa's code. Conversely, when Alyssa constructs a number for general use, she tags it with a type so that it can be appropriately recognized by the higher-level functions. This discipline of stripping off and attaching tags as data objects are passed from level to level can be an important organizational strategy, as we shall see in section **??**.

2.2.3 Data-Directed Programming and Additivity

The general strategy of checking the type of a datum and calling an appropriate function is called *dispatching on type*. This is a powerful strategy for obtaining modularity in system design. On the other hand, implementing the dispatch as in section 2.2.2 has two significant weaknesses. One weakness is that the generic interface functions (`real_part`, `imag_part`, `magnitude`, and `angle`) must know about all the different representations. For instance, suppose we wanted to incorporate a new representation for complex numbers into our complex-number system. We would need to identify this new representation with a type, and then add a clause to each of the generic interface functions to check for the new type and apply the appropriate selector for that representation.

Another weakness of the technique is that even though the individual representations can be designed separately, we must guarantee that no two functions in the entire system have the same name. This is why Ben and Alyssa had to change the names of their original functions from section 2.2.1.

The issue underlying both of these weaknesses is that the technique for implementing generic interfaces is not *additive*. The person implementing the generic selector functions must modify those functions each time a new representation is installed, and the people interfacing the individual representations must modify their code to avoid name conflicts. In each of these cases, the changes that must be made to the code are straightforward, but they must be made nonetheless, and this is a source of inconvenience and error. This is not much of a problem for the complex-number system as it stands, but suppose there were not two but hundreds of different representations for complex numbers. And suppose that there were many generic selectors to be maintained in the abstract-data interface. Suppose, in fact, that no one programmer knew all the interface functions or all the representations. The problem is real and must be addressed in such programs as large-scale data-base-management systems.

What we need is a means for modularizing the system design even further. This is provided by the programming technique known as *data-directed programming*. To understand how data-directed programming works, begin with the observation that whenever we deal with a set of generic operations that are common to a set of different types we are, in effect, dealing with a two-dimensional table that contains the possible operations on one axis and the possible types on the other axis. The entries in the table are the functions that implement each operation for each type of argument presented. In the complex-number system developed in the previous section, the correspondence between operation name, data type, and actual function was spread out among the various conditional clauses in the generic interface functions. But the same information could have been organized in a table, as shown in figure 2.18.

Data-directed programming is the technique of designing programs to work with such a table

directly. Previously, we implemented the mechanism that interfaces the complex-arithmetic code with the two representation packages as a set of functions that each perform an explicit dispatch on type. Here we will implement the interface as a single function that looks up the combination of the operation name and argument type in the table to find the correct function to apply, and then applies it to the contents of the argument. If we do this, then to add a new representation package to the system we need not change any existing functions; we need only add new entries to the table.

	Types	
Operations	Polar	Rectangular
`real-part`	`real-part-polar`	`real-part-rectangular`
`imag-part`	`imag-part-polar`	`imag-part-rectangular`
`magnitude`	`magnitude-polar`	`magnitude-rectangular`
`angle`	`angle-polar`	`angle-rectangular`

Figure 2.18: Table of operations for the complex-number system.

To implement this plan, assume that we have two functions, `put` and `get`, for manipulating the operation-and-type table:

- `put`(⟨*op*⟩, ⟨*type*⟩, ⟨*item*⟩) installs the ⟨*item*⟩ in the table, indexed by the ⟨*op*⟩ and the ⟨*type*⟩.
- `get`(⟨*op*⟩, ⟨*type*⟩) looks up the ⟨*op*⟩, ⟨*type*⟩ entry in the table and returns the item found there. If no item is found, `get` returns `undefined`.

For now, we can assume that `put` and `get` are included in our language. In chapter 3 (section 3.3.3) we will see how to implement these and other operations for manipulating tables.

Here is how data-directed programming can be used in the complex-number system. Ben, who developed the rectangular representation, implements his code just as he did originally. He defines a collection of functions, or a *package*, and interfaces these to the rest of the system by adding entries to the table that tell the system how to operate on rectangular numbers. This is accomplished by calling the following function:

```
function install_rectangular_package() {
    function real_part(z) { return head(z); }
    function imag_part(z) { return tail(z); }
    function make_from_real_imag(x, y) { return pair(x, y); }
    function magnitude(z) {
        return math_sqrt(square(real_part(z)) +
```

```
                    square(imag_part(z)));
    }
    function angle(z) {
        return math_atan(imag_part(z), real_part(z));
    }
    function make_from_mag_ang(r, a) {
        return pair(r * math_cos(a), r * math_sin(a));
    }
    // interface to the rest of the system
    function tag(x) {
        return attach_tag("rectangular", x);
    }
    put("real_part", list("rectangular"), real_part);
    put("imag_part", list("rectangular"), imag_part);
    put("magnitude", list("rectangular"), magnitude);
    put("angle", list("rectangular"), angle);
    put("make_from_real_imag", "rectangular",
        (x, y) => tag(make_from_real_imag(x, y)));
    put("make_from_mag_ang", "rectangular",
        (r, a) => tag(make_from_mag_ang(r, a)));
    return "done";
}

install_rectangular_package();
```

Notice that the internal functions here are the same functions from section 2.2.1 that Ben wrote when he was working in isolation. No changes are necessary in order to interface them to the rest of the system. Moreover, since these function definitions are internal to the installation function, Ben needn't worry about name conflicts with other functions outside the rectangular package. To interface these to the rest of the system, Ben installs his `real_part` function under the operation name `real_part` and the type `list("rectangular")`, and similarly for the other selectors.[27] The interface also defines the constructors to be used by the external system.[28] These are identical to Ben's internally defined constructors, except that they attach the tag. Alyssa's polar package is analogous:

```
function install_polar_package() {
    // internal functions
    function magnitude(z) { return head(z); }
    function angle(z) { return tail(z); }
    function make_from_mag_ang(r, a) { return pair(r, a); }
    function real_part(z) {
       return magnitude(z) * math_cos(angle(z));
    }
```

[27] We use the list `list("rectangular")` rather than the string `"rectangular"` to allow for the possibility of operations with multiple arguments, not all of the same type.

[28] 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.

```
    function imag_part(z) {
       return magnitude(z) * math_sin(angle(z));
    }
    function make_from_real_imag(x, y) {
       return pair(math_sqrt(square(x) + square(y)),
                   math_atan(y, x));
    }

    // interface to the rest of the system
    function tag(x) { return attach_tag("polar", x); }
    put("real_part", list("polar"), real_part);
    put("imag_part", list("polar"), imag_part);
    put("magnitude", list("polar"), magnitude);
    put("angle", list("polar"), angle);
    put("make_from_real_imag", "polar",
        (x, y) => tag(make_from_real_imag(x, y)));
    put("make_from_mag_ang", "polar",
        (r, a) => tag(make_from_mag_ang(r, a)));
    return "done";
}

install_polar_package();
```

Even though Ben and Alyssa both still use their original functions defined with the same names as each other's (e.g., `real_part`), these definitions are now internal to different functions (see section **??**), so there is no name conflict.

The complex-arithmetic selectors access the table by means of a general 'operation' function called `apply_generic`, which applies a generic operation to some arguments. The function `apply_generic` looks in the table under the name of the operation and the types of the arguments and applies the resulting function if one is present:[29]

```
function apply_generic(op, args) {
    const type_tags = map(type_tag, args);
    const fun = get(op, type_tags);
    return fun !== undefined
        ? apply(fun, map(contents, args))
        : Error("No method for these types in apply_generic",
                list(op, type_tags));
}
```

[29] 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. The function `apply_generic` also uses the primitive function `apply`, which takes two arguments, a function and a list. The function `apply` applies the function, using the elements in the list as arguments. For example,

```
apply(sum_of_squares, list(1, 3))
```

returns 10.

Using apply_generic , we can define our generic selectors as follows:

```
function real_part(z) {
   return apply_generic("real_part", list(z));
}
function imag_part(z) {
   return apply_generic("imag_part", list(z));
}
function magnitude(z) {
   return apply_generic("magnitude", list(z));
}
function angle(z) {
   return apply_generic("angle", list(z));
}
```

Observe that these do not change at all if a new representation is added to the system.

We can also extract from the table the constructors to be used by the programs external to the packages in making complex numbers from real and imaginary parts and from magnitudes and angles. As in section 2.2.2, we construct rectangular numbers whenever we have real and imaginary parts, and polar numbers whenever we have magnitudes and angles:

```
function make_from_real_imag(x, y) {
   return get("make_from_real_imag", "rectangular")(x, y);
}
function make_from_mag_ang(r, a) {
   return get("make_from_mag_ang", "polar")(r, a);
}
```

Exercise 2.37

Section **??** described a program that performs symbolic differentiation:

```
function deriv(exp, variable) {
    return is_number(exp)
           ? 0
           : is_variable(exp)
             ? (is_same_variable(exp, variable)) ? 1 : 0
             : is_sum(exp)
               ? make_sum(deriv(addend(exp), variable),
                          deriv(augend(exp), variable))
               : is_product(exp)
                 ? make_sum(make_product(multiplier(exp),
                                         deriv(multiplicand(exp),
                                               variable)),
                            make_product(deriv(multiplier(exp),
                                               variable),
                                         multiplicand(exp)))
```

```
                // more rules can be added here
                : Error("unknown expression type in deriv",
                        exp);
}
```

```
deriv(list("*", list("*", "x", "y"), list("+", "x", 4)), "x");
// [ "+",
//   [["*", [["*", ["x", ["y", null]]],
//           [["+", [1, [0, null]]], null]]],
//    [["*",
//      [["+",
//        [["*", ["x", [0, null]]],
//         [["*", [1, ["y", null]]], null]]],
//         [["+", ["x", [4, null]]], null] ] ],
//     null ]]]
```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the 'type tag' of the datum is the algebraic operator symbol (such as +) and the operation being performed is `deriv`. We can transform this program into data-directed style by rewriting the basic derivative function as

```
function deriv(exp, variable) {
    return is_number(exp)
           ? 0
           : is_variable(exp)
             ? (is_same_variable(exp, variable) ? 1 : 0)
             : get("deriv",
                   operator(exp))(operands(exp), variable);
}
function operator(exp) {
   return head(exp);
}
function operands(exp) {
   return tail(exp);
}
```

a. Explain what was done above. Why can't we assimilate the predicates `is_number` and `is_same_variable` into the data-directed dispatch?

b. Write the functions for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.

c. Choose any additional differentiation rule that you like, such as the one for exponents (exercise **??**), and install it in this data-directed system.

d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the functions in the opposite way, so that the dispatch line in `deriv` looked like

```
get(operator(exp), "deriv")(operands(exp), variable);
```

What corresponding changes to the derivative system are required?

Solution

Exercise 2.38

Insatiable Enterprises, Inc., is a highly decentralized conglomerate company consisting of a large number of independent divisions located all over the world. The company's computer facilities have just been interconnected by means of a clever network-interfacing scheme that makes the entire network appear to any user to be a single computer. Insatiable's president, in her first attempt to exploit the ability of the network to extract administrative information from division files, is dismayed to discover that, although all the division files have been implemented as data structures in JavaScript, the particular data structure used varies from division to division. A meeting of division managers is hastily called to search for a strategy to integrate the files that will satisfy headquarters' needs while preserving the existing autonomy of the divisions. Show how such a strategy can be implemented with data-directed programming. As an example, suppose that each division's personnel records consist of a single file, which contains a set of records keyed on employees' names. The structure of the set varies from division to division. Furthermore, each employee's record is itself a set (structured differently from division to division) that contains information keyed under identifiers such as `address` and `salary`. In particular:

a. Implement for headquarters a `get_record` function that retrieves a specified employee's record from a specified personnel file. The function should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?

b. Implement for headquarters a `get_salary` function that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?

c. Implement for headquarters a `find_employee_record` function. This should search all the divisions' files for the record of a given employee and return the record. Assume that this function takes as arguments an employee's name and a list of all the divisions' files.

d. When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?

Solution

Message passing

The key idea of data-directed programming is to handle generic operations in programs by dealing explicitly with operation-and-type tables, such as the table in figure 2.18. The style of programming we used in section 2.2.2 organized the required dispatching on type by having each operation take care of its own dispatching. In effect, this decomposes the operation-and-type table into rows, with each generic operation function representing a row of the table.

An alternative implementation strategy is to decompose the table into columns and, instead of using 'intelligent operations' that dispatch on data types, to work with 'intelligent data objects' that dispatch on operation names. We can do this by arranging things so that a data object, such as a rectangular number, is represented as a function that takes as input the required operation name and performs the operation indicated. In such a discipline, `make_from_real_imag` could be written as

```
function make_from_real_imag(x, y) {
    function dispatch(op) {
        return op === "real_part"
            ? x
            : op === "imag_part"
              ? y
              : op === "magnitude"
                ? math_sqrt(square(x) + square(y))
                : op === "angle"
                  ? math_atan(y, x)
                  : Error("Unknown op in make_from_real_imag",
                          op);
    }
    return dispatch;
}
```

The corresponding `apply_generic` function, which applies a generic operation to an argument, now simply feeds the operation's name to the data object and lets the object do the work:[30]

```
function apply_generic(op, arg) {
    return head(arg)(op);
}
```

[30] One limitation of this organization is it permits only generic functions of one argument.

Note that the value returned by `make_from_real_imag` is a function—the internal `dispatch` function. This is the function that is invoked when `apply_generic` requests an operation to be performed.

This style of programming is called *message passing*. The name comes from the image that a data object is an entity that receives the requested operation name as a 'message.' We have already seen an example of message passing in section **??**, where we saw how `pair`, `head`, and `tail` could be defined with no data objects but only functions. Here we see that message passing is not a mathematical trick but a useful technique for organizing systems with generic operations. In the remainder of this chapter we will continue to use data-directed programming, rather than message passing, to discuss generic arithmetic operations. In chapter 3 we will return to message passing, and we will see that it can be a powerful tool for structuring simulation programs.

Exercise 2.39

Implement the constructor `make_from_mag_ang` in message-passing style. This function should be analogous to the `make_from_real_imag` function given above. Solution

Exercise 2.40

As a large system with generic operations evolves, new types of data objects or new operations may be needed. For each of the three strategies—generic operations with explicit dispatch, data-directed style, and message-passing-style—describe the changes that must be made to a system in order to add new types or new operations. Which organization would be most appropriate for a system in which new types must often be added? Which would be most appropriate for a system in which new operations must often be added? Solution

Chapter 3

Modularity, Objects, and State

Μεταβάλλον – ναπαύεται (Even while it changes, it stands still.)

— Heraclitus

Plus Ça change, plus c'est la même chose.

— Alphonse Karr

The preceding chapters introduced the basic elements from which programs are made. We saw how primitive functions and primitive data are combined to construct compound entities, and we learned that abstraction is vital in helping us to cope with the complexity of large systems. But these tools are not sufficient for designing programs. Effective program synthesis also requires organizational principles that can guide us in formulating the overall design of a program. In particular, we need strategies to help us structure large systems so that they will be *modular*, that is, so that they can be divided 'naturally' into coherent parts that can be separately developed and maintained.

One powerful design strategy, which is particularly appropriate to the construction of programs for modeling physical systems, is to base the structure of our programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

To a large extent, then, the way we organize a large program is dictated by our perception of the system to be modeled. In this chapter we will investigate two prominent organizational strategies arising from two rather different 'world views' of the structure of systems. The first organizational strategy concentrates on *objects*, viewing a large system as a collection of distinct objects whose behaviors may change over time. An alternative organizational strategy concentrates on the *streams* of information that flow in the system, much as an electrical engineer views a signal-processing system.

Both the object-based approach and the stream-processing approach raise significant linguistic issues in programming. With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to abandon our old substitution model of computation (section **??**) in favor of a more mechanistic but less theoretically tractable *environment model* of computation. The difficulties of dealing with objects, change, and identity are a fundamental consequence of the need to grapple with time in our computational models. These difficulties become even greater when we allow the possibility of concurrent execution of programs. The stream approach can be most fully exploited when we decouple simulated time in our model from the order of the events that take place in the computer during evaluation. We will accomplish this using a technique known as *delayed evaluation*.

3.1 Assignment and Local State

We ordinarily view the world as populated by independent objects, each of which has a state that changes over time. An object is said to 'have state' if its behavior is influenced by its history. A bank account, for example, has state in that the answer to the question 'Can I withdraw $100?' depends upon the history of deposit and withdrawal transactions. We can characterize an object's state by one or more *state variables*, which among them maintain enough information about history to determine the object's current behavior. In a simple banking system, we could characterize the state of an account by a current balance rather than by remembering the entire history of account transactions.

In a system composed of many objects, the objects are rarely completely independent. Each may influence the states of others through interactions, which serve to couple the state variables of one object to those of other objects. Indeed, the view that a system is composed of separate objects is most useful when the state variables of the system can be grouped into closely coupled subsystems that are only loosely coupled to other subsystems.

This view of a system can be a powerful framework for organizing computational models of the system. For such a model to be modular, it should be decomposed into computational objects that model the actual objects in the system. Each computational object must have its own *local state variables* describing the actual object's state. Since the states of objects in the

system being modeled change over time, the state variables of the corresponding computational objects must also change. If we choose to model the flow of time in the system by the elapsed time in the computer, then we must have a way to construct computational objects whose behaviors change as our programs run. In particular, if we wish to model state variables by ordinary symbolic names in the programming language, then the language must provide an *assignment operator* to enable us to change the value associated with a name.

3.1.1 Local State Variables

To illustrate what we mean by having a computational object with time-varying state, let us model the situation of withdrawing money from a bank account. We will do this using a function `withdraw`, which takes as argument an `amount` to be withdrawn. If there is enough money in the account to accommodate the withdrawal, then `withdraw` should return the balance remaining after the withdrawal. Otherwise, `withdraw` should return the message *Insufficient funds.* For example, if we begin with $100 in the account, we should obtain the following sequence of responses using `withdraw`:

```
withdraw(25); // output: 75
```

```
withdraw(25); // output: 50
```

```
withdraw(60); // output: "Insufficient funds"
```

```
withdraw(15); // output: 35
```

Observe that the expression `withdraw(25)`, evaluated twice, yields different values. This is a new kind of behavior for a function. Until now, all our functions could be viewed as specifications for computing mathematical functions. A call to a function computed the value of the function applied to the given arguments, and two calls to the same function with the same arguments always produced the same result.[1]

So far, all our names were *constants* as declared by the keyword **`const`**. Once declared, they did not change their value, as appropriate for constants. To implement functions like `withdraw`, we introduce a new kind of declaration—*variable declarations* using the keyword **`let`** instead of **`const`**. After declaring a variable `balance`, to indicate the balance of money in the account, we can define `withdraw` as a function that accesses `balance`. The `withdraw` function checks to see if `balance` is at least as large as the requested `amount`. If so, `withdraw` decrements `balance` by `amount` and returns the new value of `balance`. Otherwise, `withdraw` returns the *Insufficient funds* message. Here are the declarations of `balance` and `withdraw`:

[1] Actually, this is not quite true. One exception was the random-number generator in section **??**. Another exception involved the operation/type tables we introduced in section 2.2.3, where the values of two calls to `get` with the same arguments depended on intervening calls to `put`. On the other hand, until we introduce assignment, we have no way to create such functions ourselves.

```
let balance = 100;

function withdraw(amount) {
   if (balance >= amount) {
      balance = balance - amount;
      return balance;
   } else {
      return "Insufficient funds";
   }
}
```

Decrementing balance is accomplished by the statement

```
balance = balance - amount;
```

The syntax of such *assignment statements* is

```
name = new-value;
```

Here *name* is a symbol and *new-value* is any expression. The assignment changes *name* so that its value is the result obtained by evaluating *new-value*. In the case at hand, we are changing balance so that its new value will be the result of subtracting amount from the previous value of balance. [2]

The function withdraw also uses a *sequential composition* to cause two expressions to be evaluated in the case where the **if** test is true: first decrementing balance and then returning the value of balance. In general, executing the statement

```
stmt1 stmt2
```

causes the statements $stmt_1$ and $stmt_2$ to be evaluated in sequence.[3]

Although withdraw works as desired, the variable balance presents a problem. As specified above, balance is a name defined in the program environment and is freely accessible to be examined or modified by any function. It would be much better if we could somehow make balance internal to withdraw, so that withdraw would be the only function that could access

[2] Note that assignment statements look similar to and should not be confused with constant and variable declarations of the form

```
const name = value;
```

and

```
let name = value;
```

in which a newly declared *name* is associated with a *value*. Also similar in looks but not in meaning are expressions of the form

```
expression1 === expression2
```

which evaluate to true if $expression_1$ evaluates to the same value as $expression_2$ and to false, otherwise.

[3] We have already used sequential composition implicitly in our programs, because in JavaScript the body of a function can be a sequence of statements, not just a single **return** statement, as discussed in section **??**.

balance directly and any other function could access balance only indirectly (through calls to withdraw). This would more accurately model the notion that balance is a local state variable used by withdraw to keep track of the state of the account.

We can make balance internal to withdraw by rewriting the definition as follows:

```
function make_withdraw() {
    let balance = 100;
    return amount => {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "insufficient funds";
        }
    };
}
const new_withdraw = make_withdraw();
```

What we have done here is use **let** to establish an environment with a local variable balance, bound to the initial value 100. Within this local environment, we use function definition[4] to create a function that takes amount as an argument and behaves like our previous withdraw function. This function—returned as the result of evaluating the body of the make_withdraw function—behaves in precisely the same way as withdraw but whose variable balance is not accessible by any other function.[5]

Combining assignment statements with variable statements is the general programming technique we will use for constructing computational objects with local state. Unfortunately, using this technique raises a serious problem: When we first introduced functions, we also introduced the substitution model of evaluation (section **??**) to provide an interpretation of what function application means. We said that applying a function should be interpreted as evaluating the body of the function with the formal parameters replaced by their values. The trouble is that, as soon as we introduce assignment into our language, substitution is no longer an adequate model of function application. (We will see why this is so in section 3.1.3.) As a consequence, we technically have at this point no way to understand why the new_withdraw function behaves as claimed above. In order to really understand a function such as new_withdraw, we will need to develop a new model of function application. In section 3.2 we will introduce such a model, together with an explanation of assignment statements and variable statements. First, however, we examine some variations on the theme established by make_withdraw.

[4]Blocks as bodies of function definition expressions were introduced in section 2.1.4.

[5]In programming-language jargon, the variable balance is said to be *encapsulated* within the new_withdraw function. Encapsulation reflects the general system-design principle known as the *hiding principle*: One can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a 'need to know.'

The following function, make_withdraw_with_balance, abstracts the initial balance into a parameter. The formal parameter balance in make_withdraw_with_balance specifies the initial amount of money in the account.[6]

```
function make_withdraw_with_balance(balance) {
    return amount => {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "insufficient funds";
        }
    };
}
```

The function make_withdraw_with_balance can be used as follows to create two objects w1 and w2:

```
const w1 = make_withdraw_with_balance(100);
const w2 = make_withdraw_with_balance(100);
```

```
w1(50); // output: 50
```

```
w2(70); // output: 30
```

```
w2(40); // output: "Insufficient funds"
```

```
w1(40); // output: 10
```

Observe that w1 and w2 are completely independent objects, each with its own local state variable balance. Withdrawals from one do not affect the other.

We can also create objects that handle deposits as well as withdrawals, and thus we can represent simple bank accounts. Here is a function that returns a 'bank-account object' with a specified initial balance:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
```

[6] In contrast with make_withdraw above, we do not have to use **let** to make balance a local variable, since formal parameters are already local. This will be clearer after the discussion of the environment model of evaluation in section 3.2. (See also exercise 3.10.)

```
        balance = balance + amount;
        return balance;
    }
    function dispatch(m) {
        if (m === "withdraw") {
            return withdraw;
        } else if (m === "deposit") {
            return deposit;
        } else {
            return "Unknown request - - MAKE-ACCOUNT";
        }
    }
    return dispatch;
}
```

Each call to make_account sets up an environment with a local state variable balance. Within this environment, make_account defines functions deposit and withdraw that access balance and an additional function dispatch that takes a 'message' as input and returns one of the two local functions. The dispatch function itself is returned as the value that represents the bank-account object. This is precisely the *message-passing* style of programming that we saw in section 2.2.3, although here we are using it in conjunction with the ability to modify local variables.

make_account can be used as follows:

```
const acc = make_account(100);
```

```
(acc("withdraw"))(50);
```

```
(acc("withdraw"))(60);
```

```
(acc("deposit"))(40);
```

```
(acc("withdraw"))(60);
```

Each call to acc returns the locally defined deposit or withdraw function, which is then applied to the specified amount. As was the case with make_withdraw, another call to make_account

```
const acc2 = make_account(100);
```

will produce a completely separate account object, which maintains its own local balance.

Exercise 3.1

An *accumulator* is a function that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a function make_accumulator that generates accumulators, each maintaining an

independent sum. The input to `make_accumulator` should specify the initial value of the sum; for example

```
// make_accumulator to be written by students
const a = make_accumulator(5);
```

```
a(10);  // output: 15
```

```
a(10);  // output: 25
```

Solution

Exercise 3.2

In software-testing applications, it is useful to be able to count the number of times a given function is called during the course of a computation. Write a function `make_monitored` that takes as input a function, `f`, that itself takes one input. The result returned by `make_monitored` is a third function, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the string `"how many calls?"`, then `mf` returns the value of the counter. If the input is the string `"reset count"`, then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `sqrt` function:

```
// make_monitored function to be written by students
const s = make_monitored(math_sqrt);
```

```
s(100);
```

```
s("how many calls?"); // returns 1
```

Solution

Exercise 3.3

Modify the `make_account` function so that it creates password-protected accounts. That is, `make_account` should take a symbol as an additional argument, as in

```
// make_account function to be written by students
const acc = make_account(100, "secret password");
```

The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

```
(acc("secret password", "withdraw"))(40); // result: 60
```

```
(acc("some other password", "deposit"))(40);
// result: incorrect password
```

Solution

Exercise 3.4

Modify the `make_account` function of exercise 3.3 by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the function `call_the_cops`. Solution

3.1.2 The Benefits of Introducing Assignment

As we shall see, introducing assignment into our programming language leads us into a thicket of difficult conceptual issues. Nevertheless, viewing systems as collections of objects with local state is a powerful technique for maintaining a modular design. As a simple example, consider the design of a function `rand` that, whenever it is called, returns an integer chosen at random.

It is not at all clear what is meant by 'chosen at random.' What we presumably want is for successive calls to `rand` to produce a sequence of numbers that has statistical properties of uniform distribution. We will not discuss methods for generating suitable sequences here. Rather, let us assume that we have a function `rand_update` that has the property that if we start with a given number `x_1` and form

```
x_2 = rand_update(x_1);
x_3 = rand_update(x_2);
```

then the sequence of values `x_1`, `x_2`, `x_3`, ..., will have the desired statistical properties.[7]

We can implement `rand` as a function with a local state variable `x` that is initialized to some fixed value `random_init`. Each call to `rand` computes `rand_update` of the current value of `x`, returns this as the random number, and also stores this as the new value of `x`.

[7] One common way to implement `rand_update` is to use the rule that x is updated to $ax + b$ modulo m, where a, b, and m are appropriately chosen integers. Chapter 3 of Knuth 1981 includes an extensive discussion of techniques for generating sequences of random numbers and establishing their statistical properties. Notice that the `rand_update` function computes a mathematical function: Given the same input twice, it produces the same output. Therefore, the number sequence produced by `rand_update` certainly is not 'random,' if by 'random' we insist that each number in the sequence is unrelated to the preceding number. The relation between 'real randomness' and so-called *pseudo-random* sequences, which are produced by well-determined computations and yet have suitable statistical properties, is a complex question involving difficult issues in mathematics and philosophy. Kolmogorov, Solomonoff, and Chaitin have made great progress in clarifying these issues; a discussion can be found in Chaitin 1975.

```
function make_rand() {
   let x = random_init;
   function rand() {
        x = rand_update(x);
        return x;
   }
   return rand;
}
const rand = make_rand();
```

Of course, we could generate the same sequence of random numbers without using assignment by simply calling `rand_update` directly. However, this would mean that any part of our program that used random numbers would have to explicitly remember the current value of `x` to be passed as an argument to `rand_update`. To realize what an annoyance this would be, consider using random numbers to implement a technique called *Monte Carlo simulation.*

The Monte Carlo method consists of choosing sample experiments at random from a large set and then making deductions on the basis of the probabilities estimated from tabulating the results of those experiments. For example, we can approximate π using the fact that $6/\pi^2$ is the probability that two integers chosen at random will have no factors in common; that is, that their greatest common divisor will be 1.[8] To obtain the approximation to π, we perform a large number of experiments. In each experiment we choose two integers at random and perform a test to see if their GCD is 1. The fraction of times that the test is passed gives us our estimate of $6/\pi^2$, and from this we obtain our approximation to π.

The heart of our program is a function `monte_carlo`, which takes as arguments the number of times to try an experiment, together with the experiment, represented as a no-argument function that will return either true or false each time it is run. `monte_carlo` runs the experiment for the designated number of trials and returns a number telling the fraction of the trials in which the experiment was found to be true.

```
function estimate_pi(trials) {
    return math_sqrt(6 / monte_carlo(trials, cesaro_test));
}

function cesaro_test() {
    return gcd(rand(), rand()) === 1;
}
function monte_carlo(trials, experiment) {
    function iter(trials_remaining, trials_passed) {
        if (trials_remaining === 0) {
            return trials_passed / trials;
        } else if (experiment()) {
            return iter(trials_remaining - 1,
```

[8] This theorem is due to E. Cesàro. See section 4.5.2 of Knuth 1981 for a discussion and a proof.

```
                        trials_passed + 1);
        } else {
            return iter(trials_remaining - 1,
                        trials_passed);
        }
    }
    return iter(trials, 0);
}
```

Now let us try the same computation using `rand_update` directly rather than `rand`, the way we would be forced to proceed if we did not use assignment to model local state:

```
function estimate_pi(trials) {
    return math_sqrt(6 / random_gcd_test(trials, random_init));
}

function random_gcd_test(trials, initial_x) {
    function iter(trials_remaining, trials_passed, x) {
        const x1 = rand_update(x);
        const x2 = rand_update(x1);
        if (trials_remaining === 0) {
            return trials_passed / trials;
        } else if (gcd(x1, x2) === 1) {
            return iter(trials_remaining - 1,
                        trials_passed + 1, x2);
        } else {
            return iter(trials_remaining - 1,
                        trials_passed, x2);
        }
    }
    return iter(trials, 0, initial_x);
}
```

While the program is still simple, it betrays some painful breaches of modularity. In our first version of the program, using `rand`, we can express the Monte Carlo method directly as a general `monte_carlo` function that takes as an argument an arbitrary `experiment` function. In our second version of the program, with no local state for the random-number generator, `random_gcd_test` must explicitly manipulate the random numbers `x1` and `x2` and recycle `x2` through the iterative loop as the new input to `rand_update`. This explicit handling of the random numbers intertwines the structure of accumulating test results with the fact that our particular experiment uses two random numbers, whereas other Monte Carlo experiments might use one random number or three. Even the top-level function `estimate_pi` has to be concerned with supplying an initial random number. The fact that the random-number generator's insides are leaking out into other parts of the program makes it difficult for us to isolate the Monte Carlo idea so that it can be applied to other tasks. In the first version of the

program, assignment encapsulates the state of the random-number generator within the `rand` function, so that the details of random-number generation remain independent of the rest of the program.

The general phenomenon illustrated by the Monte Carlo example is this: From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write computer programs whose structure reflects this decomposition, we make computational objects (such as bank accounts and random-number generators) whose behavior changes with time. We model state with local state variables, and we model the changes of state with assignments to those variables.

It is tempting to conclude this discussion by saying that, by introducing assignment and the technique of hiding state in local variables, we are able to structure systems in a more modular fashion than if all state had to be manipulated explicitly, by passing additional parameters. Unfortunately, as we shall see, the story is not so simple.

Exercise 3.5

Monte Carlo integration is a method of estimating definite integrals by means of Monte Carlo simulation. Consider computing the area of a region of space described by a predicate $P(x, y)$ that is true for points (x, y) in the region and false for points not in the region. For example, the region contained within a circle of radius 3 centered at $(5, 7)$ is described by the predicate that tests whether $(x-5)^2 + (y-7)^2 \leq 3^2$. To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at $(2, 4)$ and $(8, 10)$ contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region. We can estimate the integral by picking, at random, points (x, y) that lie in the rectangle, and testing $P(x, y)$ for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

Implement Monte Carlo integration as a function `estimate_integral` that takes as arguments a predicate `P`, upper and lower bounds `x1`, `x2`, `y1`, and `y2` for the rectangle, and the number of trials to perform in order to produce the estimate. Your function should use the same `monte_carlo` function that was used above to estimate π. Use your `estimate_integral` to produce an estimate of π by measuring the area of a unit circle.

You will find it useful to have a function that returns a number chosen at random from a given range. The following `random_in_range` function implements this in terms of the `random` function used in section **??**, which returns a nonnegative number less than its input.

```
function random_in_range(low, high) {
```

```
    const range = high - low;
    return low + random(range);
}
```

Exercise 3.6

It is useful to be able to reset a random-number generator to produce a sequence starting from a given value. Design a new `rand` function that is called with an argument that is either the symbol `generate` or the symbol `reset` and behaves as follows: `rand("generate")` produces a new random number; `(rand("reset"))(`*new-value*`)` resets the internal state variable to the designated *new-value*. Thus, by resetting the state, one can generate repeatable sequences. These are very handy to have when testing and debugging programs that use random numbers. Solution

3.1.3 The Costs of Introducing Assignment

As we have seen, the assignment statement enables us to model objects that have local state. However, this advantage comes at a price. Our programming language can no longer be interpreted in terms of the substitution model of function application that we introduced in section **??**. Moreover, no simple model with 'nice' mathematical properties can be an adequate framework for dealing with objects and assignment in programming languages.

So long as we do not use assignments, two evaluations of the same function with the same arguments will produce the same result, so that functions can be viewed as computing mathematical functions. Programming without any use of assignments, as we did throughout the first two chapters of this book, is accordingly known as *functional programming*.

To understand how assignment complicates matters, consider a simplified version of the `make_withdraw` function of section 3.1.1 that does not bother to check for an insufficient amount:

```
function make_simplified_withdraw(balance) {
    return amount => {
               balance = balance - amount;
               return balance;
           };
}
```

Compare this function with the following `make_decrementer` function, which does not use assignment:

```
function make_decrementer(balance) {
    return amount => balance - amount;
}
```

The function `make_decrementer` returns a function that subtracts its input from a designated amount `balance`, but there is no accumulated effect over successive calls, as with `make_simplified_withdraw`:

```
const d = make_decrementer(25);
```

```
d(20); // output: 5
```

```
d(10); // output: 15
```

We can use the substitution model to explain how `make_decrementer` works. For instance, let us analyze the evaluation of the expression

```
(make_decrementer(25))(20);
```

We first simplify the operator of the combination by substituting 25 for `balance` in the body of `make-decrementer`. This reduces the expression to

```
(amount => 25 - amount)(20);
```

Now we apply the operator by substituting 20 for `amount` in the body of the function definition expression:

```
25 - 20;
```

The final answer is 5.

Observe, however, what happens if we attempt a similar substitution analysis with `make_simplified_withdraw`

```
(make_simplified_withdraw(25))(20);
```

We first simplify the operator by substituting 25 for `balance` in the return expression of `make_simplified_withdraw`. This reduces the expression to[9]

```
(amount => {
     balance = 25 - amount;
     return 25;
 })(20);
```

Now we apply the function by substituting 20 for `amount` in the body of the function:

```
balance = 25 - 20;
return 25;
```

If we adhered to the substitution model, we would have to say that the meaning of the function

[9] We don't substitute for the occurrence of `balance` in the assignment statement because the name in an assignment is not evaluated. If we did substitute for it, we would get `25 = 25 - amount;`, which makes no sense.

application is to first set `balance` to 5 and then return 25 as the value of the expression. This gets the wrong answer. In order to get the correct answer, we would have to somehow distinguish the first occurrence of `balance` (before the effect of the assignment) from the second occurrence of `balance` (after the effect of the assignment), and the substitution model cannot do this.

The trouble here is that substitution is based ultimately on the notion that the symbols in our language are essentially names for values. But as soon as we introduce assignment and the idea that the value of a variable can change, a variable can no longer be simply a name. Now a variable somehow refers to a place where a value can be stored, and the value stored at this place can change. In section 3.2 we will see how environments play this role of 'place' in our computational model.

Sameness and change

The issue surfacing here is more profound than the mere breakdown of a particular model of computation. As soon as we introduce change into our computational models, many notions that were previously straightforward become problematical. Consider the concept of two things being 'the same.'

Suppose we call `make_decrementer` twice with the same argument to create two functions:

```
const d1 = make_decrementer(25);

const d2 = make_decrementer(25);
```

Are `d1` and `d2` the same? An acceptable answer is yes, because `d1` and `d2` have the same computational behavior—each is a function that subtracts its input from 25. In fact, `d1` could be substituted for `d2` in any computation without changing the result.

Contrast this with making two calls to `make_simplified_withdraw`:

```
const w1 = make_simplified_withdraw(25);

const w2 = make_simplified_withdraw(25);
```

Are `w1` and `w2` the same? Surely not, because calls to `w1` and `w2` have distinct effects, as shown by the following sequence of interactions:

```
w1(20);

w1(20);

w2(20);
```

Even though `w1` and `w2` are 'equal' in the sense that they are both created by evaluating the same expression, `make_simplified_withdraw(25)`, it is not true that `w1` could be substituted for `w2` in any expression without changing the result of evaluating the expression.

A language that supports the concept that 'equals can be substituted for equals' in an expression without changing the value of the expression is said to be *referentially transparent*. Referential transparency is violated when we include assignment in our computer language. This makes it tricky to determine when we can simplify expressions by substituting equivalent expressions. Consequently, reasoning about programs that use assignment becomes drastically more difficult.

Once we forgo referential transparency, the notion of what it means for computational objects to be 'the same' becomes difficult to capture in a formal way. Indeed, the meaning of 'same' in the real world that our programs model is hardly clear in itself. In general, we can determine that two apparently identical objects are indeed 'the same one' only by modifying one object and then observing whether the other object has changed in the same way. But how can we tell if an object has 'changed' other than by observing the 'same' object twice and seeing whether some property of the object differs from one observation to the next? Thus, we cannot determine 'change' without some *a priori* notion of 'sameness,' and we cannot determine sameness without observing the effects of change.

As an example of how this issue arises in programming, consider the situation where Peter and Paul have a bank account with $100 in it. There is a substantial difference between modeling this as

```
const peter_acc = make_account(100);
const paul_acc = make_account(100);
```

and modeling it as

```
const peter_acc = make_account(100);
const paul_acc = peter_acc;
```

In the first situation, the two bank accounts are distinct. Transactions made by Peter will not affect Paul's account, and vice versa. In the second situation, however, we have defined `paul_acc` to be *the same thing* as `peter_acc` In effect, Peter and Paul now have a joint bank account, and if Peter makes a withdrawal from `peter_acc` Paul will observe less money in `paul_acc`. These two similar but distinct situations can cause confusion in building computational models. With the shared account, in particular, it can be especially confusing that there is one object (the bank account) that has two different names (`peter_acc` and `paul_acc`); if we are searching for all the places in our program where `paul_acc` can be changed, we must remember to look also at things that change `peter_acc`.[10]

[10]The phenomenon of a single computational object being accessed by more than one name is known as *aliasing*. The joint bank account situation illustrates a very simple example of an alias. In section 3.3 we will see much more complex examples, such as 'distinct' compound data structures that share parts. Bugs can occur in our programs if we forget that a change to an object may also, as a 'side effect,' change a 'different' object because the two 'different' objects are actually a single object appearing under different aliases. These so-called *side-effect bugs* are so difficult to locate and to analyze that some people have proposed that programming languages be designed in such a way as to not allow side effects or aliasing (Lampson et al. 1981; Morris, Schmidt, and Wadler 1980).

With reference to the above remarks on 'sameness' and 'change,' observe that if Peter and Paul could only examine their bank balances, and could not perform operations that changed the balance, then the issue of whether the two accounts are distinct would be moot. In general, so long as we never modify data objects, we can regard a compound data object to be precisely the totality of its pieces. For example, a rational number is determined by giving its numerator and its denominator. But this view is no longer valid in the presence of change, where a compound data object has an 'identity' that is something different from the pieces of which it is composed. A bank account is still 'the same' bank account even if we change the balance by making a withdrawal; conversely, we could have two different bank accounts with the same state information. This complication is a consequence, not of our programming language, but of our perception of a bank account as an object. We do not, for example, ordinarily regard a rational number as a changeable object with identity, such that we could change the numerator and still have 'the same' rational number.

Pitfalls of imperative programming

In contrast to functional programming, programming that makes extensive use of assignment is known as *imperative programming*. In addition to raising complications about computational models, programs written in imperative style are susceptible to bugs that cannot occur in functional programs. For example, recall the iterative factorial program from section **??**:

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

Instead of passing arguments in the internal iterative loop, we could adopt a more imperative style by using explicit assignment to update the values of the variables `product` and `counter`:

```
function factorial(n) {
   let product = 1;
   let counter = 1;
   function iter() {
      if (counter > n) {
          return product;
      } else {
          product = counter * product;
```

```
            counter = counter + 1;
            return iter();
        }
    }
    return iter();
}
```

This does not change the results produced by the program, but it does introduce a subtle trap. How do we decide the order of the assignments? As it happens, the program is correct as written. But writing the assignments in the opposite order

```
counter = counter + 1;
product = counter * product;
```

would have produced a different, incorrect result. In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs.[11]

The complexity of imperative programs becomes even worse if we consider applications in which several processes execute concurrently. We will return to this in section 3.4. First, however, we will address the issue of providing a computational model for expressions that involve assignment, and explore the uses of objects with local state in designing simulations.

Exercise 3.7

Consider the bank account objects created by `make_account`, with the password modification described in exercise 3.3. Suppose that our banking system requires the ability to make joint accounts. Define a function `make_joint` that accomplishes this. The function `make_joint` should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the `make_joint` operation to proceed. The third argument is a new password. The function `make_joint` is to create an additional access to the original account using the new password. For example, if (`peter_acc` is a bank account with password (`"open sesame"`, then

```
// make_joint function to be written by students
const paul_acc =
    make_joint(peter_acc, "open sesame", "rosebud");
```

will allow one to make transactions on (`peter_acc` using the name `paul_acc` and the password

[11] In view of this, it is ironic that introductory programming is most often taught in a highly imperative style. This may be a vestige of a belief, common throughout the 1960s and 1970s, that programs that call functions must inherently be less efficient than programs that perform assignments. (Steele (1977) debunks this argument.) Alternatively it may reflect a view that step-by-step assignment is easier for beginners to visualize than function call. Whatever the reason, it often saddles beginning programmers with 'should I set this variable before or after that one' concerns that can complicate programming and obscure the important ideas.

`"rosebud"`. You may wish to modify your solution to exercise 3.3 to accommodate this new feature. Solution

Exercise 3.8

When we defined the evaluation model in section ??, we said that the first step in evaluating an expression is to evaluate its subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a function are evaluated can make a difference to the result. Define a simple function `f` such that evaluating `f(0) + f(1)` will return 0 if the arguments to + are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

3.2 The Environment Model of Evaluation

When we introduced compound functions in chapter 1, we used the substitution model of evaluation (section ??) to define what is meant by applying a function to arguments:

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

Once we admit assignment into our programming language, such a definition is no longer adequate. In particular, section 3.1.3 argued that, in the presence of assignment, a variable can no longer be considered to be merely a name for a value. Rather, a variable must somehow designate a 'place' in which values can be stored. In our new model of evaluation, these places will be maintained in structures called *environments*.

An environment is a sequence of *frames*. Each frame is a table (possibly empty) of *bindings*, which associate variable names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its *enclosing environment*, unless, for the purposes of discussion, the frame is considered to be *global*. The *value of a variable* with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be *unbound* in the environment.

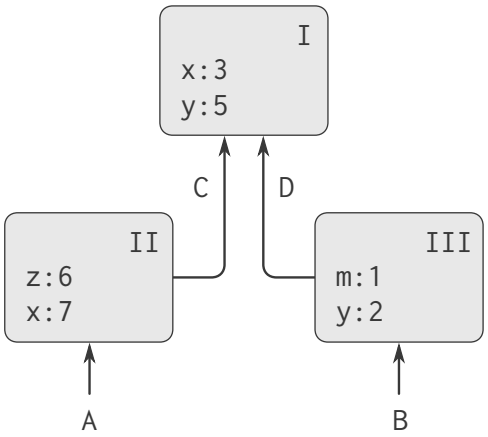


Figure 3.1: A simple environment structure.

Figure 3.1 shows a simple environment structure consisting of three frames, labeled I, II, and III. In the diagram, A, B, C, and D are pointers to environments. C and D point to the same environment. The variables z and x are bound in frame II, while y and x are bound in frame I. The value of x in environment D is 3. The value of x with respect to environment B is also 3. This is determined as follows: We examine the first frame in the sequence (frame III) and do not find a binding for x, so we proceed to the enclosing environment D and find the binding in frame I. On the other hand, the value of x in environment A is 7, because the first frame in the sequence (frame II) contains a binding of x to 7. With respect to environment A, the binding of x to 7 in frame II is said to *shadow* the binding of x to 3 in frame I.

The environment is crucial to the evaluation process, because it determines the context in which an expression should be evaluated. Indeed, one could say that expressions in a programming language do not, in themselves, have any meaning. Rather, an expression acquires a meaning only with respect to some environment in which it is evaluated. Even the interpretation of an expression as straightforward as `display(1)` depends on an understanding that one is operating in a context in which `display` refers to the primitive function that displays a value. Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment. To describe interactions with the interpreter, we will suppose that there is a program environment, consisting of a single frame (directly inside the global environment) that includes values for the symbols associated with the primitive functions. For example, the idea that `display` is the primitive function for displaying a value is captured by saying that the variable `display` is bound in the global environment to the respective primitive function.

3.2.1 The Rules for Evaluation

The overall specification of how the interpreter evaluates an application combination remains the same as when we first introduced it in section **??**:

- To evaluate an application combination of the form

  `function-expression ( argument-expressions )`

  do the following:

  - Evaluate the function expression of the application combination, resulting in the function to be applied.
  - Evaluate the argument expressions of the combination.
  - Apply the function to the arguments.

The environment model of evaluation replaces the substitution model in specifying what it means to apply a compound function to arguments.

In the environment model of evaluation, a function is always a pair consisting of some code and a pointer to an environment. Functions are created in one way only: by evaluating a function definition expression. This produces a function whose code is obtained from the text of the function definition expression and whose environment is the environment in which the function definition expression was evaluated to produce the function. For example, consider the function declaration

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

evaluated in the global environment. The function declaration syntax is just syntactic sugar for an underlying implicit function definition expression. It would have been equivalent to have used

```
const square = x => x * x;
```

which evaluates `x => x * x` and binds `square` to the resulting value, all in the global environment.

Figure 3.2 shows the result of evaluating this function declaration statement. The function object is a pair whose code specifies that the function has one formal parameter, namely `x`, and a function body `return x * x;`. The environment part of the function is a pointer to the program environment, since that is the environment in which the function definition expression was evaluated to produce the function. A new binding, which associates the function object with the symbol `square`, has been added to the program frame. The bindings in a frame correspond

to the const and let declarations directly nested in that frame. The program gives rise to a frame of its own, which we call the *program environment*, directly inside the global frame. To reduce clutter, after this figure, we will not display the global environment (as it is always the same), but we are reminded of its existence by the pointer from the program environment.

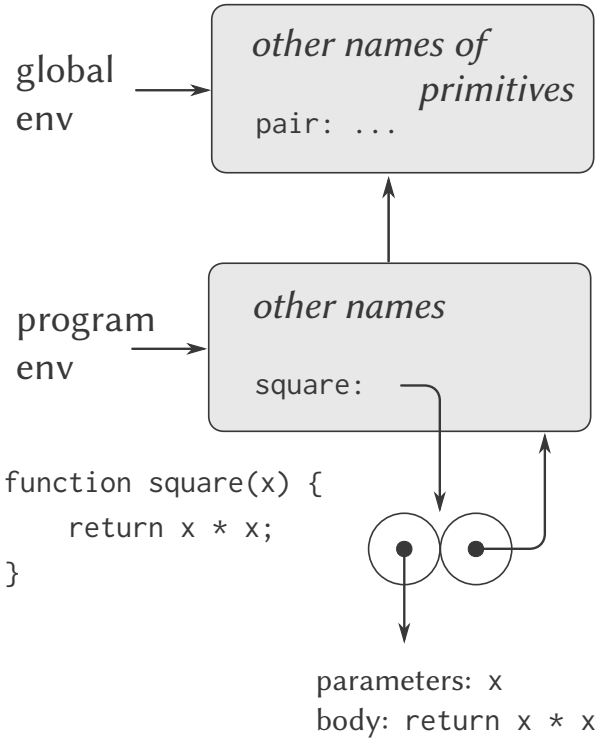


Figure 3.2: Environment structure produced by evaluating `function square(x) return x * x;` in the program environment.

Now that we have seen how functions are created, we can describe how functions are applied. The environment model specifies: To apply a function to arguments, create a new environment containing a frame that binds the parameters to the values of the arguments. The enclosing environment of this frame is the environment specified by the function. Now, within this new environment, evaluate the function body.

To show how this rule is followed, Figure 3.3 illustrates the environment structure created by evaluating the expression `square(5);` in the program environment, where `square` is the function generated in Figure 3.2. Applying the function results in the creation of a new environment, labeled E1 in the figure, that begins with a frame in which `x`, the formal parameter for the function, is bound to the argument 5. The pointer leading upward from this frame shows that the frame's enclosing environment is the program environment. The program environment is chosen here, because this is the environment that is indicated as part of the `square` function object. Within E1, we evaluate the body of the function, `return x * x;`. Since the value of `x` in E1 is 5, the result is `5 * 5`, or 25.

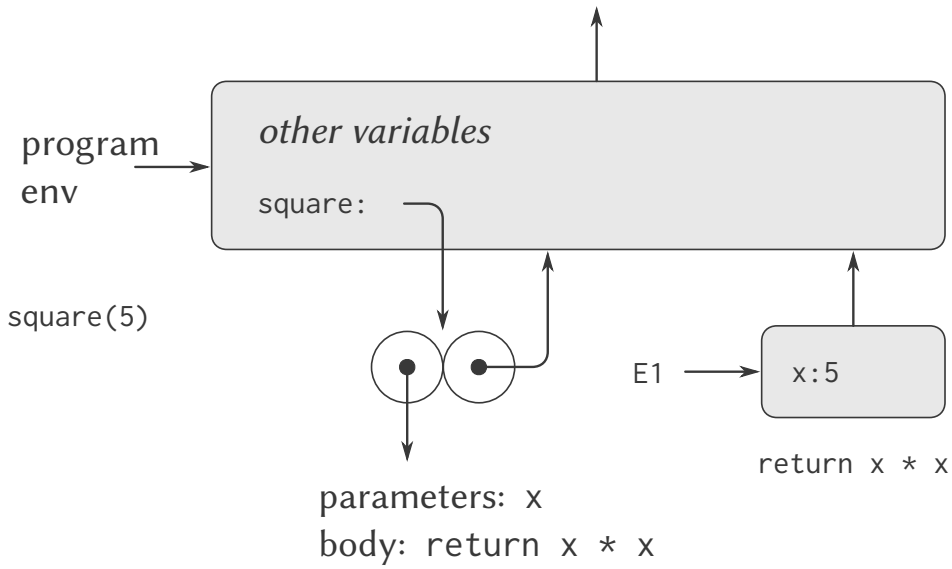


Figure 3.3: Environment created by evaluating `square(5);` in the program environment.

The environment model of function application can be summarized by two rules:

- A function object is applied to a set of arguments by constructing a frame, in which we create variable bindings of the parameters of the function to the arguments of the call, and then evaluating the body of the function in the context of the new environment constructed. The new frame has as its enclosing environment the environment part of the function object being applied.
- A function is created by evaluating a function definition expression relative to a given environment. The resulting function object is a pair consisting of the text of the function definition expression and a pointer to the environment in which the function was created.

We also specify that defining a symbol using **`const`**/**`let`** creates a constant/variable binding in the current environment frame and assigns to the symbol the indicated value. Finally, we specify the behavior of assignment, the operation that forced us to introduce the environment model in the first place. Evaluating the statement *name* = *value*; in some environment locates the binding of the name in the environment. For this, one finds the first frame in the environment that contains a binding for the name. If the name is unbound in the environment, then the assignment signals a 'variable undefined' error. Otherwise, if the binding in the frame is a constant binding, the assignment signals an 'assignment to constant' error, because JavaScript forbids assignment to constants. At last, if the binding in the frame is a variable binding, that binding is changed to reflect the new value of the variable.

These evaluation rules, though considerably more complex than the substitution model, are still reasonably straightforward. Moreover, the evaluation model, though abstract, provides a

correct description of how the interpreter evaluates expressions. In chapter 4 we shall see how this model can serve as a blueprint for implementing a working interpreter. The following sections elaborate the details of the model by analyzing some illustrative programs.

3.2.2 Applying Simple Functions

When we introduced the substitution model in section ?? we showed how the combination `f(5)` evaluates to 136, given the following function definitions:

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

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

We can analyze the same example using the environment model. Figure 3.4 shows the three function objects created by evaluating the definitions of `f`, `square`, and `sum_of_squares` in the program environment. Each function object consists of some code, together with a pointer to the program environment.

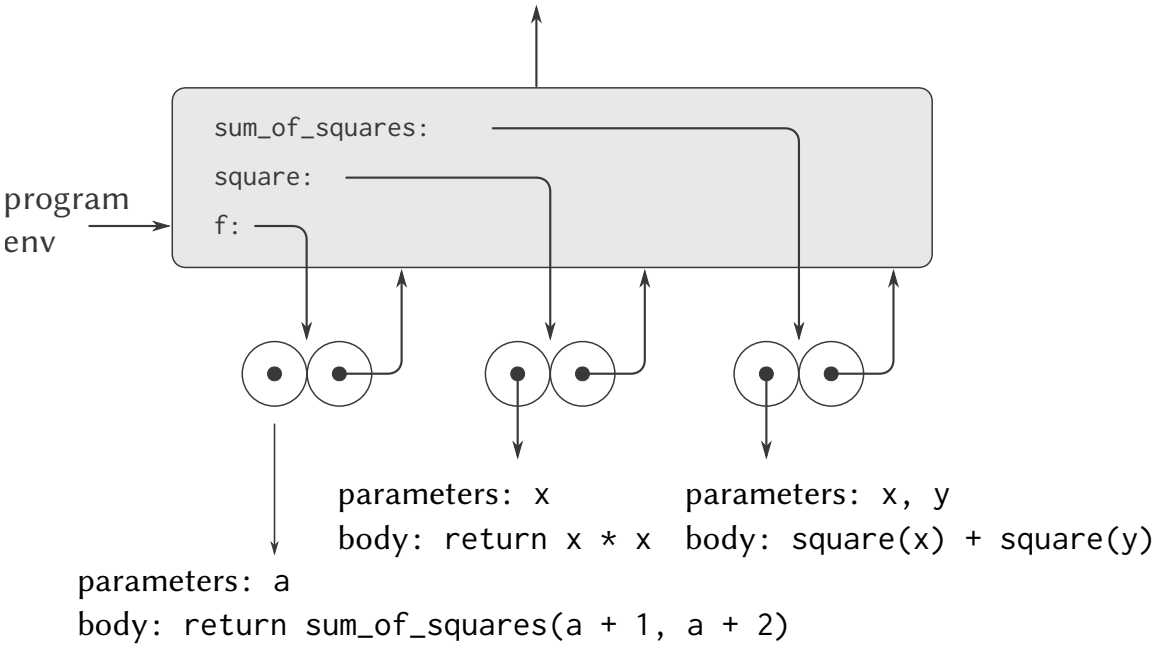


Figure 3.4: Function objects in the program frame.

In Figure 3.5 we see the environment structure created by evaluating the expression `f(5)`. The call to `f` creates a new environment E1 beginning with a frame in which `a`, the formal parameter of `f`, is bound to the argument 5. In E1, we evaluate the body of `f`:

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

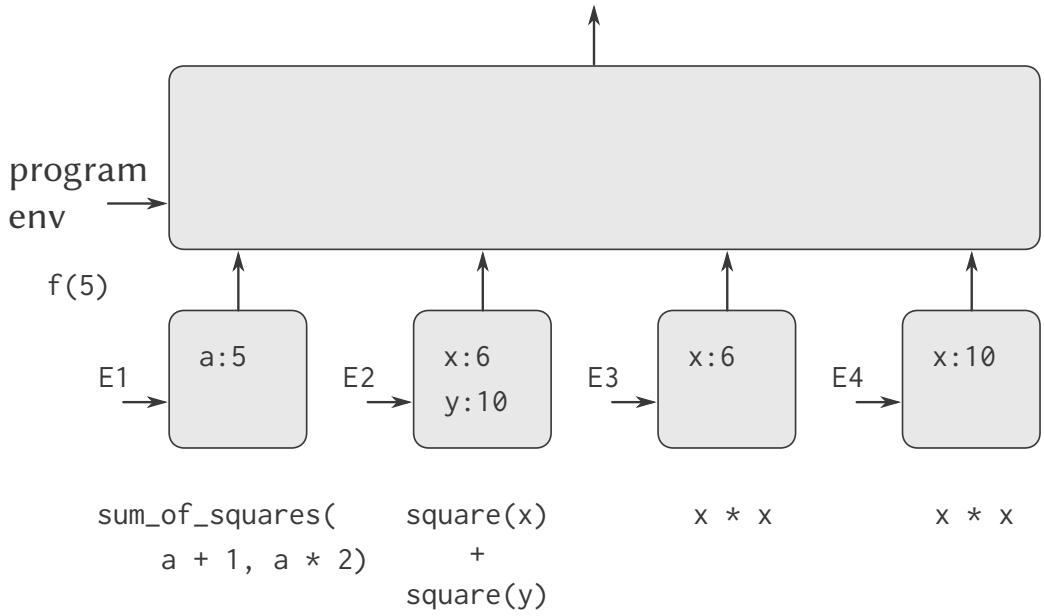


Figure 3.5: Environments created by evaluating `f(5)` using the functions in Figure 3.4.

To evaluate this combination, we first evaluate the subexpressions. The first subexpression, `sum_of_squares`, has a value that is a function object. (Notice how this value is found: We first look in the first frame of E1, which contains no binding for `sum_of_squares`. Then we proceed to the enclosing environment, i.e. the program environment, and find the binding shown in Figure 3.4.) The other two subexpressions are evaluated by applying the primitive operations `+` and `*` to evaluate the two combinations `a + 1` and `a * 2` to obtain 6 and 10, respectively.

Now we apply the function object `sum_of_squares` to the arguments 6 and 10. This results in a new environment E2 in which the formal parameters `x` and `y` are bound to the arguments. Within E2 we evaluate the combination `square(x) + square(y)`. This leads us to evaluate `square(x)`, where `square` is found in the program frame and `x` is 6. Once again, we set up a new environment, E3, in which `x` is bound to 6, and within this we evaluate the body of `square`, which is `x * x`. Also as part of applying `sum_of_squares`, we must evaluate the subexpression `square(y)`, where `y` is 10. This second call to `square` creates another environment, E4, in which `x`, the formal parameter of `square`, is bound to 10. And within E4 we must evaluate `x * x`.

The important point to observe is that each call to `square` creates a new environment containing a binding for `x`. We can see here how the different frames serve to keep separate the

different local variables all named `x`. Notice that each frame created by `square` points to the program environment, since this is the environment indicated by the `square` function object.

After the subexpressions are evaluated, the results are returned. The values generated by the two calls to `square` are added by `sum_of_squares`, and this result is returned by `f`. Since our focus here is on the environment structures, we will not dwell on how these returned values are passed from call to call; however, this is also an important aspect of the evaluation process, and we will return to it in detail in chapter 5.

Exercise 3.9

In section ?? we used the substitution model to analyze two functions for computing factorials, a recursive version

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

and an iterative version

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

Show the environment structures created by evaluating `factorial(6)` using each version of the `factorial` function. [12]

3.2.3 Frames as the Repository of Local State

We can turn to the environment model to see how functions and assignment can be used to represent objects with local state. As an example, consider the 'withdrawal processor' from section 3.1.1 created by calling the function

12

```
function make_withdraw_with_balance(balance) {
    return amount => {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "insufficient funds";
        }
    };
}
```

Let us describe the evaluation of

```
const w1 = make_withdraw_with_balance(100);
```

followed by

```
w1(50);
```

Figure 3.6 shows the result of declaring the `make_withdraw_with_balance` function in the program environment. This produces a function object that contains a pointer to the program environment. So far, this is no different from the examples we have already seen, except that the body of the function is itself a function definition expression.

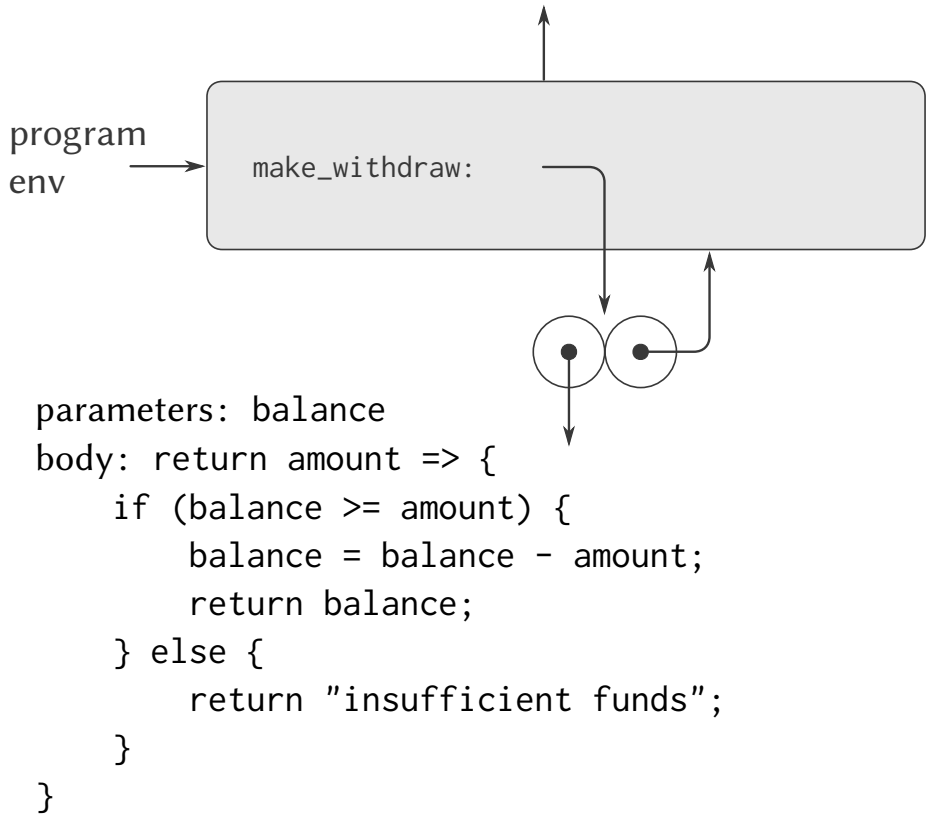


Figure 3.6: Result of defining `make_withdraw_with_balance` in the program environment.

The interesting part of the computation happens when we apply the function

```
make_withdraw_with_balance
```

to an argument:

```
const w1 = make_withdraw_with_balance(100);
```

We begin, as usual, by setting up an environment E1 in which the formal parameter `balance` is bound to the argument 100. Within this environment, we evaluate the body of `make_withdraw_with_balance`, namely the function definition expression. This constructs a new function object, whose code is as specified by the function definition and whose environment is E1, the environment in which the function definition was evaluated to produce the function. The resulting function object is the value returned by the call to `make_withdraw_with_balance`. This is bound to `w1` in the program environment, since the constant declaration itself is being evaluated in the program environment. Figure 3.7 shows the resulting environment structure.

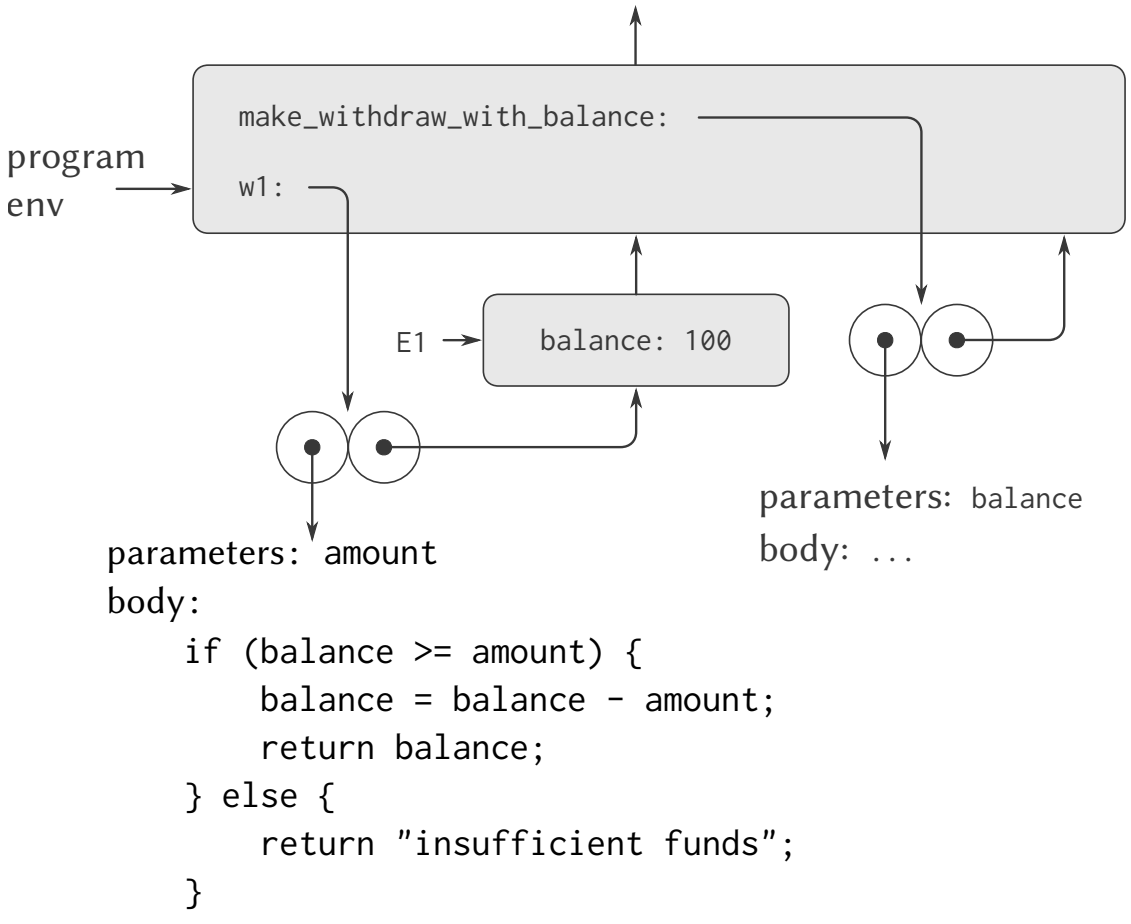


Figure 3.7: Result of evaluating **`const`** `w1 = make_withdraw_with_balance(100);`.

Now we can analyze what happens when `w1` is applied to an argument:

```
w1(50);
```

We begin by constructing a frame in which amount, the formal parameter of w1, is bound to the argument 50. The crucial point to observe is that this frame has as its enclosing environment not the program environment, but rather the environment E1, because this is the environment that is specified by the w1 function object. Within this new environment, we evaluate the body of the function:

```
if (balance >= amount) {
    balance = balance - amount;
    return balance;
} else {
    return "insufficient funds";
}
```

The resulting environment structure is shown in Figure 3.8. The expression being evaluated references both amount and balance. The variable amount will be found in the first frame in the environment, while balance will be found by following the enclosing-environment pointer to E1.

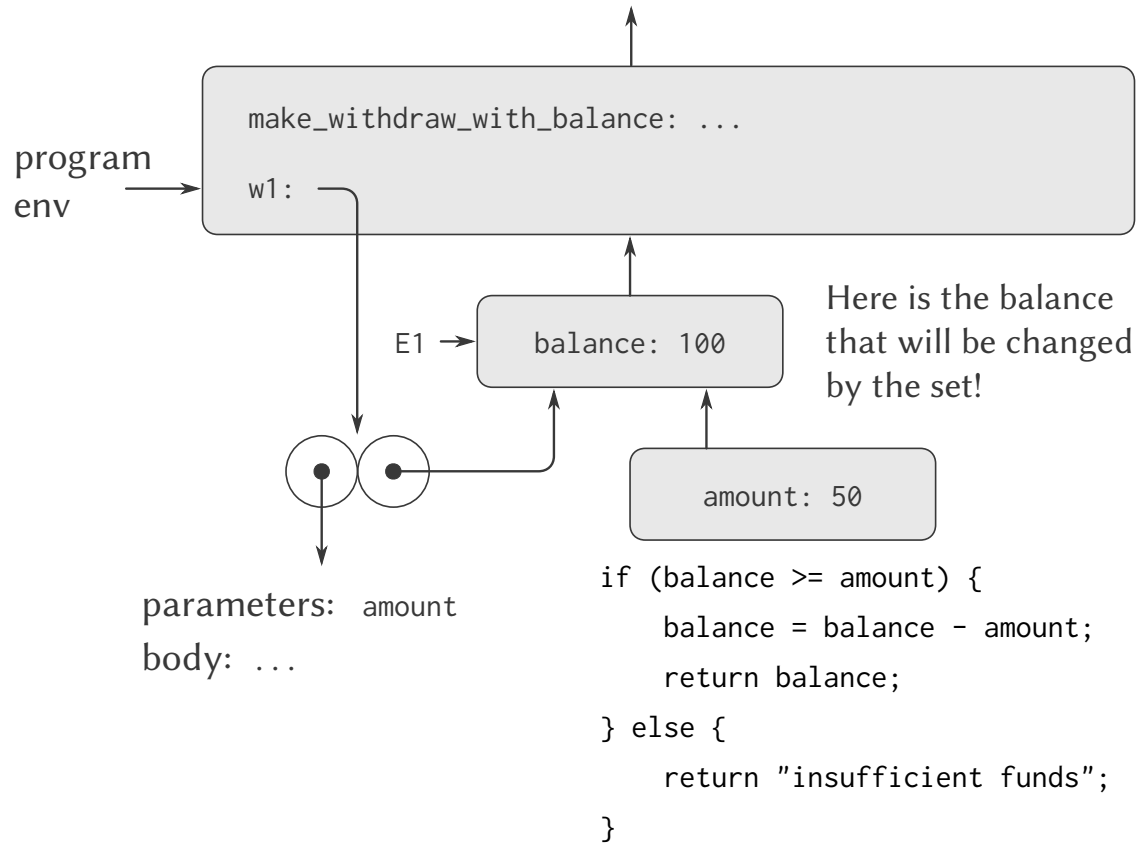


Figure 3.8: Environments created by applying the function object w1.

When the assignment is executed, the binding of balance in E1 is changed. At the completion of the call to w1, balance is 50, and the frame that contains balance is still pointed to by the function object w1. The frame that binds amount (in which we executed the code that changed

balance) is no longer relevant, since the function call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment. The next time w1 is called, this will build a new frame that binds amount and whose enclosing environment is E1. We see that E1 serves as the 'place' that holds the local state variable for the function object w1. Figure 3.9 shows the situation after the call to W1.

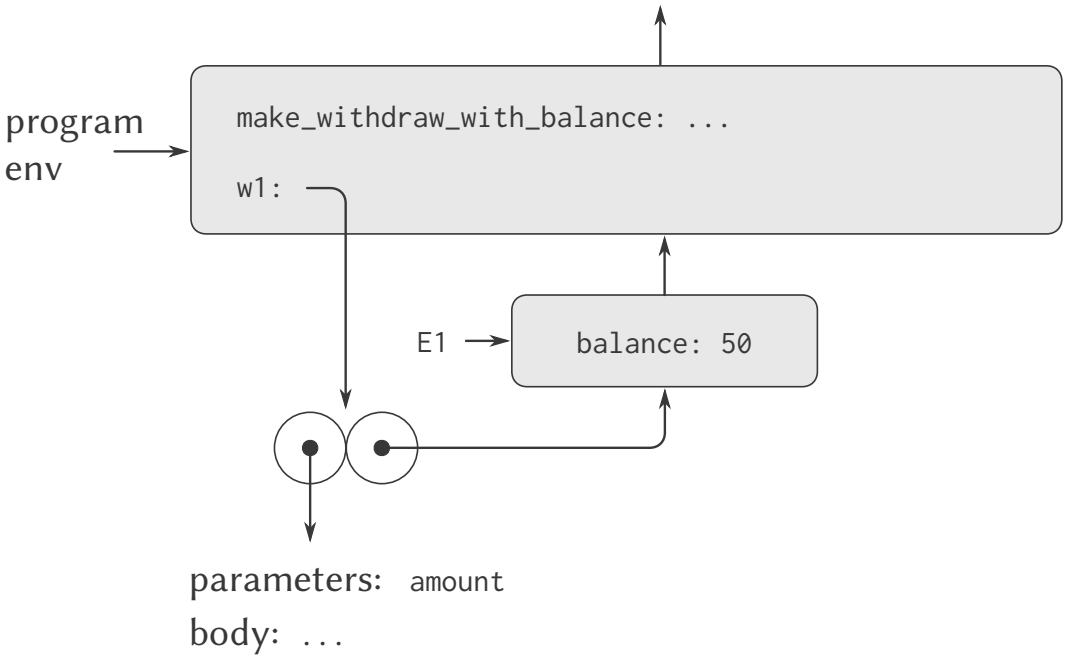


Figure 3.9: Environments after the call to w1.

Observe what happens when we create a second 'withdraw' object by making another call to make_withdraw:

```
const w2 = make_withdraw(100);
```

This produces the environment structure of Figure 3.10, which shows that w2 is a function object, that is, a pair with some code and an environment. The environment E2 for w2 was created by the call to make_withdraw. It contains a frame with its own local binding for balance. On the other hand, w1 and w2 have the same code: the code specified by the function definition expression in the body of make_withdraw.[13] We see here why w1 and w2 behave as independent objects. Calls to w1 reference the state variable balance stored in E1, whereas calls to w2 reference the balance stored in E2. Thus, changes to the local state of one object do not affect the other object.

[13]Whether w1 and w2 share the same physical code stored in the computer, or whether they each keep a copy of the code, is a detail of the implementation. For the interpreter we implement in chapter 4, the code is in fact shared.

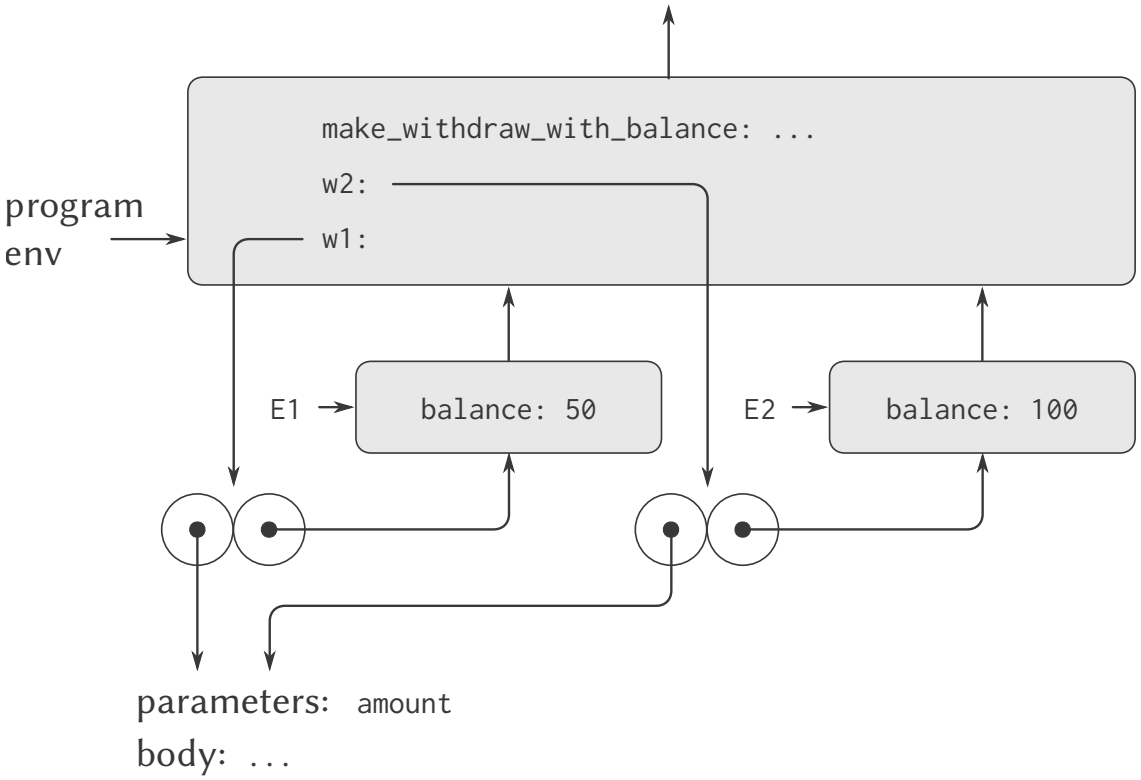


Figure 3.10: Using **const** `w2 = make_withdraw_with_balance(100)` to create a second object.

Exercise 3.10

In the `make_withdraw` function, the local variable `balance` is created as a parameter of `make_withdraw`. We could also create the local state variable explicitly, using **let**, as follows:

```
function make_withdraw(initial_amount) {
    let balance = initial_amount;
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "insufficient funds";
        }
    }
    return withdraw;
}
```

Use the environment model to analyze this alternate version of `make_withdraw`, drawing figures like the ones above to illustrate the interactions

```
const w1 = make_withdraw(100);
w1(50);
const w2 = make_withdraw(100);
```

Show that the two versions of `make_withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?

3.2.4 Internal Definitions

Section **??** introduced the idea that functions can have internal definitions, thus leading to a block structure as in the following function to compute square roots:

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

Now we can use the environment model to see why these internal definitions behave as desired. Figure 3.11 shows the point in the evaluation of the expression `sqrt(2)` where the internal function `good_enough` has been called for the first time with `guess` equal to 1.

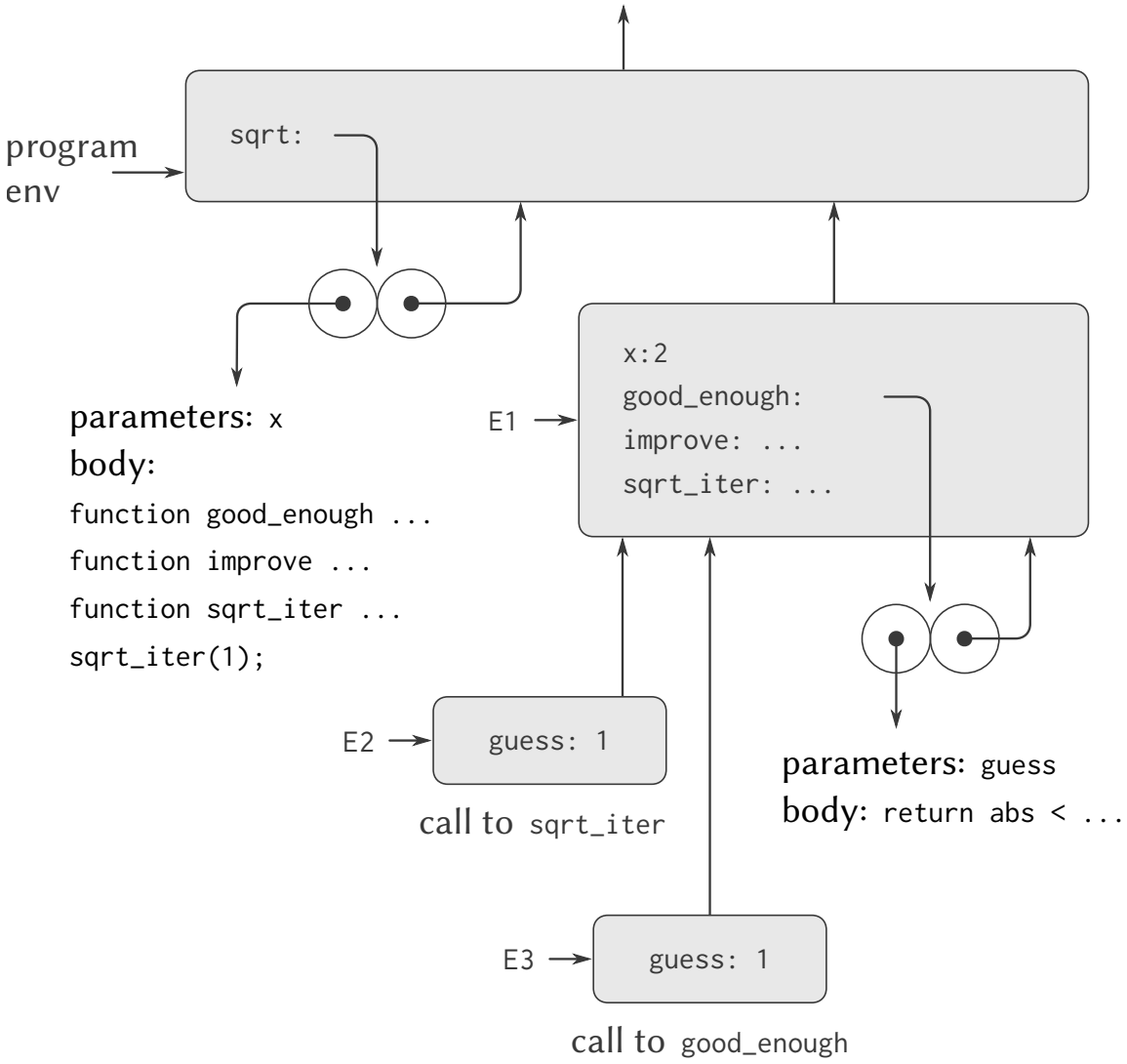


Figure 3.11: sqrt function with internal definitions.

Observe the structure of the environment. sqrt is a symbol in the program environment that is bound to a function object whose associated environment is the program environment. When sqrt was called, a new environment E1 was formed, subordinate to the program environment, in which the parameter x is bound to 2. The body of sqrt was then evaluated in E1. Since the first expression in the body of sqrt is

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

evaluating this expression defined the function good_enough in the environment E1. To be more precise, the symbol good_enough was added to the first frame of E1, bound to a function object whose associated environment is E1. Similarly, improve and sqrt_iter were defined as functions in E1. For conciseness, Figure 3.11 shows only the function object for good_enough.

After the local functions were defined, the expression `sqrt_iter(1.0)` was evaluated, still in environment E1. So the function object bound to `sqrt_iter` in E1 was called with 1 as an argument. This created an environment E2 in which `guess`, the parameter of `sqrt_iter`, is bound to 1. The function `sqrt_iter` in turn called `good_enough` with the value of `guess` (from E2) as the argument for `good_enough`. This set up another environment, E3, in which `guess` (the parameter of `good_enough`) is bound to 1. Although `sqrt_iter` and `good_enough` both have a parameter named `guess`, these are two distinct local variables located in different frames. Also, E2 and E3 both have E1 as their enclosing environment, because the `sqrt_iter` and `good_enough` functions both have E1 as their environment part. One consequence of this is that the symbol `x` that appears in the body of `good_enough` will reference the binding of `x` that appears in E1, namely the value of `x` with which the original `sqrt` function was called.

The environment model thus explains the two key properties that make local function definitions a useful technique for modularizing programs:

- The names of the local functions do not interfere with names external to the enclosing function, because the local function names will be bound in the frame that the function creates when it is run, rather than being bound in the program environment.
- The local functions can access the arguments of the enclosing function, simply by using parameter names as free variables. This is because the body of the local function is evaluated in an environment that is subordinate to the evaluation environment for the enclosing function.

Exercise 3.11

In section 3.2.3 we saw how the environment model described the behavior of functions with local state. Now we have seen how internal definitions work. A typical message-passing function contains both of these aspects. Consider the bank account function of section 3.1.1:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
    }
    function dispatch(m) {
```

```
        return m === "withdraw"
               ? withdraw
               : m === "deposit"
                 ? deposit
                 : "Unknown request: make_account";
    }
    return dispatch;
}
```

Show the environment structure generated by the sequence of interactions

```
const acc = make_account(50);
(acc("deposit"))(40);
(acc("withdraw"))(60);
```

Where is the local state for acc kept? Suppose we define another account

```
const acc2 = make_account(100);
```

How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between acc and acc2?

3.3 Modeling with Mutable Data

Note: this section is a work in progress!

Chapter 2 dealt with compound data as a means for constructing computational objects that have several parts, in order to model real-world objects that have several aspects. In that chapter we introduced the discipline of data abstraction, according to which data structures are specified in terms of constructors, which create data objects, and selectors, which access the parts of compound data objects. But we now know that there is another aspect of data that chapter 2 did not address. The desire to model systems composed of objects that have changing state leads us to the need to modify compound data objects, as well as to construct and select from them. In order to model compound objects with changing state, we will design data abstractions to include, in addition to selectors and constructors, operations called *mutators*, which modify data objects. For instance, modeling a banking system requires us to change account balances. Thus, a data structure for representing bank accounts might admit an operation

```
set_balance(account, new-value)
```

that changes the balance of the designated account to the designated new value. Data objects for which mutators are defined are known as *mutable data objects*.

Chapter 2 introduced pairs as a general-purpose 'glue' for synthesizing compound data. We begin this section by defining basic mutators for pairs, so that pairs can serve as building blocks

for constructing mutable data objects. These mutators greatly enhance the representational power of pairs, enabling us to build data structures other than the sequences and trees that we worked with in section 2.1. We also present some examples of simulations in which complex systems are modeled as collections of objects with local state.

3.3.1 Mutable List Structure

The basic operations on pairs—`pair`, `head`, and `tail`—can be used to construct list structure and to select parts from list structure, but they are incapable of modifying list structure. The same is true of the list operations we have used so far, such as `append` and `list`, since these can be defined in terms of `pair`, `head`, and `tail`. To modify list structures we need new operations.

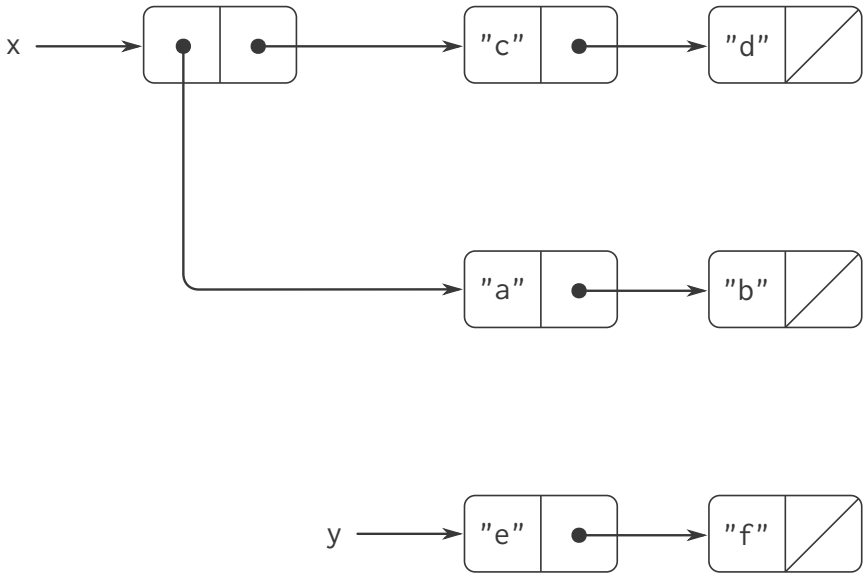


Figure 3.12: Lists `list(list("a", "b"), "c", "d")` and `y`: `list("e", "f")`.

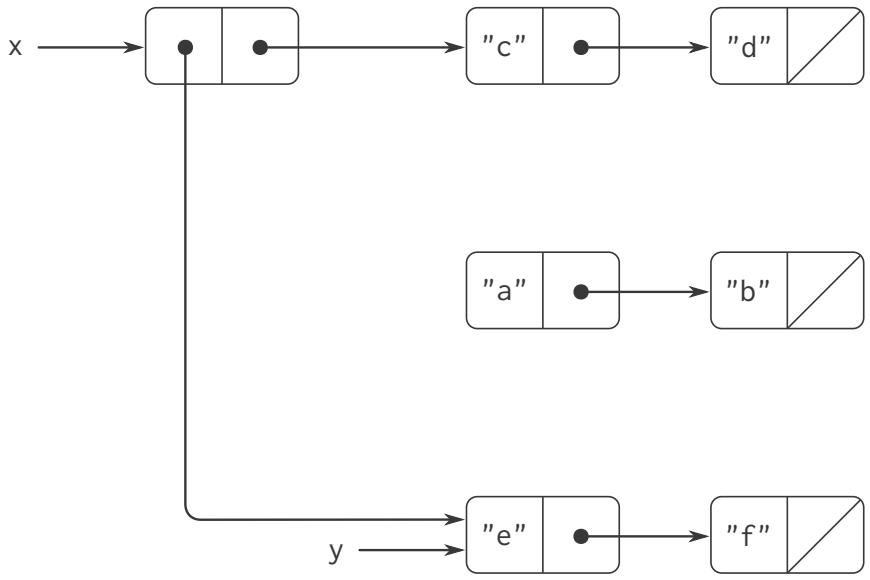


Figure 3.13: Effect of `set_head(x, y)` on the lists in figure 3.12.

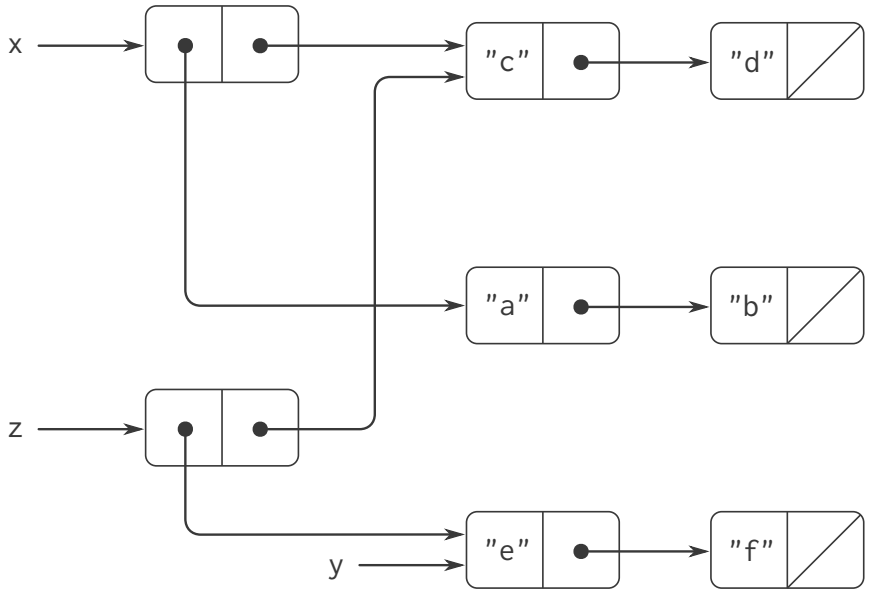


Figure 3.14: Effect of **`const`** `z = pair(y, tail(x));` on the lists in figure 3.12.

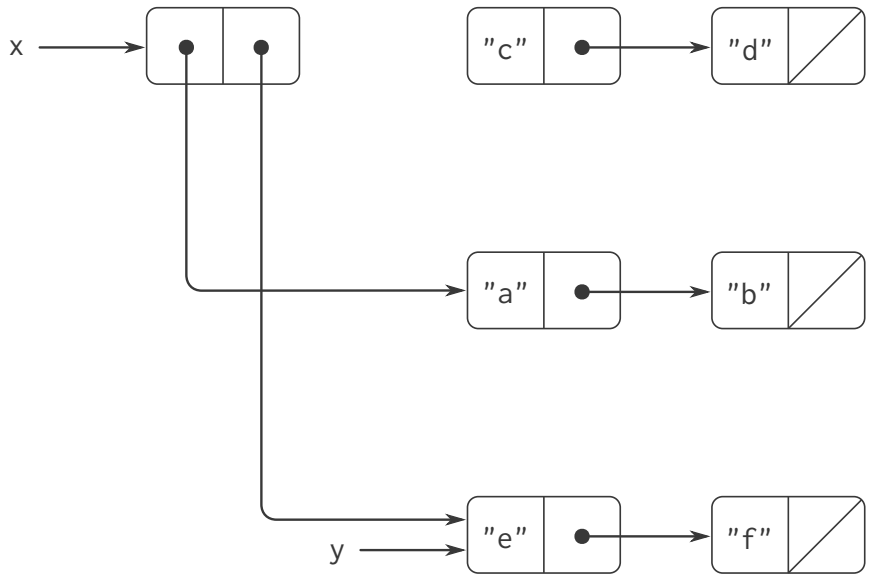


Figure 3.15: Effect of `set_tail(x, y)` on the lists in figure 3.12.

The primitive mutators for pairs are `set_head` and `set_tail`. The function `set_head` takes two arguments, the first of which must be a pair. It modifies this pair, replacing the `head` pointer by a pointer to the second argument of `set_head`. [14]

As an example, suppose that `x` is bound to the list `list(list("a", "b"), "c")` and `y` to the list `list("e", "f")` as illustrated in figure 3.12. Evaluating the expression `set_head(x, y)` modifies the pair to which `x` is bound, replacing its `head` by the value of `y`. The result of the operation is shown in figure 3.13. The structure `x` has been modified and would now be printed as `list(list("e", "f"), "c", "d")`. The pairs representing the list `list("a", "b")`, identified by the pointer that was replaced, are now detached from the original structure.[15]

Compare figure 3.13 with figure 3.14, which illustrates the result of executing

```
const z = pair(y, tail(x));
```

with `x` and `y` bound to the original lists of figure 3.12. The variable `z` is now bound to a new pair created by the `pair` operation; the list to which `x` is bound is unchanged.

The `set_tail` operation is similar to `set_head`. The only difference is that the `tail` pointer of the pair, rather than the `head` pointer, is replaced. The effect of executing `set_tail(x, y)` on the lists of figure 3.12 is shown in figure 3.15. Here the `tail` pointer of `x` has been replaced by the pointer to `list("e", "f")`. Also, the list `list("c", "d")`, which used to be the `tail` of `x`,

[14] The functions `set_head` and `set_tail` return the value `undefined`. Like assignment, they should be used only for their effect.

[15] We see from this that mutation operations on lists can create 'garbage' that is not part of any accessible structure.

is now detached from the structure.

The function pair builds new list structure by creating new pairs, while set_head and set_tail modify existing pairs. Indeed, we could implement pair in terms of the two mutators, together with a function get_new_pair, which returns a new pair that is not part of any existing list structure. We obtain the new pair, set its head and tail pointers to the designated objects, and return the new pair as the result of the pair.[16]

```
function pair(x, y) {
    const fresh = get_new_pair();
    set_head(fresh, x);
    set_tail(fresh, y);
    return fresh;
}
```

Exercise 3.12

The following function for appending lists was introduced in section 2.1.1:

```
function append(x, y) {
    return is_null(x)
           ? y
           : pair(head(x), append(tail(x), y));
}
```

The function append forms a new list by successively pairing the elements of x onto y. The function append_mutator is similar to append, but it is a mutator rather than a constructor. It appends the lists by splicing them together, modifying the final pair of x so that its tail is now y. (It is an error to call append_mutator with an empty x.)

```
function append_mutator(x, y) {
    set_tail(last_pair(x), y);
}
```

Here last_pair is a function that returns the last pair in its argument:

```
function last_pair(x) {
    return is_null(tail(x))
           ? x
           : last_pair(tail(x));
}
```

Consider the program

[16]The function get_new_pair is one of the operations that must be implemented as part of the memory management required by a JavaScript implementation.

```
const x = list("a", "b");
const y = list("c", "d");
const z = append(x, y);
display(z);       // ["a", ["b", ["c", ["d", null]]]]
display(tail(x)); // ???
const w = append_mutator(x, y);
display(w);       // ["a", ["b", ["c", ["d", null]]]]
display(tail(x)); // ???
```

What are the missing responses? Draw box-and-pointer diagrams to explain your answer.

Exercise 3.13

Consider the following `make_cycle` function, which uses the `last_pair` function defined in exercise 3.12:

```
function make_cycle(x) {
    set_tail(last_pair(x), x);
    return x;
}
```

Draw a box-and-pointer diagram that shows the structure `z` created by

```
const z = make_cycle(list("a", "b", "c"));
```

What happens if we try to compute `last_pair(z)`?

Exercise 3.14

The following function is quite useful, although obscure:

```
function mystery(x) {
    function loop(x, y) {
        if (is_null(x)) {
            return y;
        } else {
            let temp = tail(x);
            set_tail(x, y);
            return loop(temp, x);
        }
    }
    return loop(x, null);
}
```

The function `loop` uses the 'temporary' variable `temp` to hold the old value of the `tail` of `x`, since the `set_tail` on the next line destroys the `tail`. Explain what `mystery` does in general. Suppose `v` is defined by

```
const v = list("a", "b", "c");
```

Draw the box-and-pointer diagram that represents the list to which v is bound. Suppose that we now evaluate

```
const w = mystery(v);
```

Draw box-and-pointer diagrams that show the structures v and w after evaluating this program. What would be printed as the values of v and w?

Sharing and identity

We mentioned in section 3.1.3 the theoretical issues of 'sameness' and 'change' raised by the introduction of assignment. These issues arise in practice when individual pairs are *shared* among different data objects. For example, consider the structure formed by

```
const x = list("a", "b");
const z1 = pair(x, x);
```

As shown in figure 3.16, z1 is a pair whose head and tail both point to the same pair x. This sharing of x by the head and tail of z1 is a consequence of the straightforward way in which pair is implemented. In general, using pair to construct lists will result in an interlinked structure of pairs in which many individual pairs are shared by many different structures.

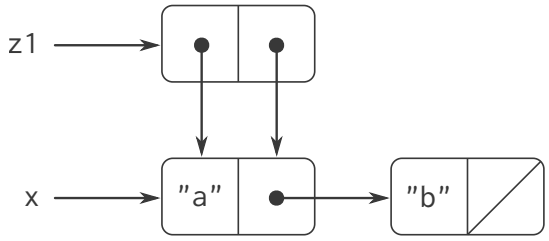


Figure 3.16: The list z1 formed by pair(x, x).

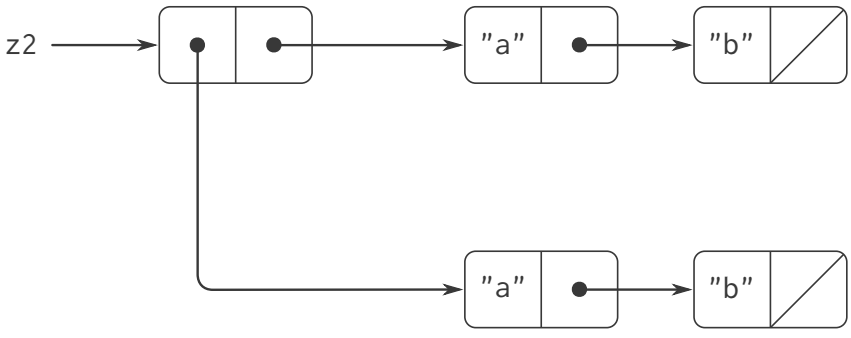


Figure 3.17: The list z2 formed by pair(list("a", "b"), list("a", "b")).

In contrast to figure 3.16, figure 3.17 shows the structure created by

```
const z2 = pair(list("a", "b"), list("a", "b"));
```

In this structure, the pairs in the two `list("a", "b")` lists are distinct, although they contain the same strings. [17]

When thought of as a list, `z1` and `z2` both represent 'the same' list:

```
list(list("a", "b"), "a", "b")
```

In general, sharing is completely undetectable if we operate on lists using only `pair`, `head`, and `tail`. However, if we allow mutators on list structure, sharing becomes significant. As an example of the difference that sharing can make, consider the following function, which modifies the `head` of the structure to which it is applied:

```
function set_to_wow(x) {
    set_head(head(x), "wow");
    return x;
}
```

Even though `z1` and `z2` are 'the same' structure, applying `set_to_wow` to them yields different results. With `z1`, altering the `head` also changes the `tail`, because in `z1` the `head` and the `tail` are the same pair. With `z2`, the `head` and `tail` are distinct, so `set-to-wow!` modifies only the `head`:

```
z1;
// displays: [["a", ["b", null], ["a", ["b", null]]
```

```
set_to_wow(z1);
// displays: [["wow", ["b", null], ["wow", ["b", null]]
```

```
z2;
// displays: [["a", ["b", null], ["a", ["b", null]]
```

```
set_to_wow(z2);
// displays: [["wow", ["b", null], ["a", ["b", null]]
```

One way to detect sharing in list structures is to use the predicate operator `===` , which we introduced in section **??** as a way to test whether two strings are equal. More generally, `x === y` tests whether `x` and `y` are the same object (that is, whether `x` and `y` are equal as pointers). Thus, with `z1` and `z2` as defined in figures 3.16 and 3.17, `head(z1) === tail(z1)` is true and `head(z2) === tail(z2)` is false.

As will be seen in the following sections, we can exploit sharing to greatly extend the repertoire of data structures that can be represented by pairs. On the other hand, sharing can also be dangerous, since modifications made to structures will also affect other structures that happen to share the modified parts. The mutation operations `set_head` and `set_tail` should be used

[17] The two pairs are distinct because each call to `pair` returns a new pair. The strings are 'the same' in the sense that they are primitive data (just like numbers) that are composed of the same characters in the same order. Since JavaScript provides no way to mutate a string, any sharing that the designers of a JavaScript interpreter might decide to implement for strings is undetectable. We consider primitive data such as numbers, booleans and strings as *identical*, if and only if they are *indistinguishable*.

with care; unless we have a good understanding of how our data objects are shared, mutation can have unanticipated results.[18]

Exercise 3.15

Draw box-and-pointer diagrams to explain the effect of `set_to_wow` on the structures `z1` and `z2` above.

Exercise 3.16

Ben Bitdiddle decides to write a function to count the number of pairs in any list structure. 'It's easy,' he reasons. 'The number of pairs in any structure is the number in the `head` plus the number in the `tail` plus one more to count the current pair.' So Ben writes the following function:

```
function count_pairs(x) {
    return !is_pair(x)
           ? 0
           : count_pairs(head(x)) +
             count_pairs(tail(x)) + 1;
}
```

Show that this function is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's function would return 3; return 4; return 7; never return at all.

Exercise 3.17

Devise a correct version of the `count_pairs` function of exercise 3.16 that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

Exercise 3.18

Write a function that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by taking successive `tails` would go into an

[18] The subtleties of dealing with sharing of mutable data objects reflect the underlying issues of 'sameness' and 'change' that were raised in section 3.1.3. We mentioned there that admitting change to our language requires that a compound object must have an 'identity' that is something different from the pieces from which it is composed. In JavaScript, we consider this 'identity' to be the quality that is tested by ===, i.e., by equality of pointers. Since in most JavaScript implementations a pointer is essentially a memory address, we are 'solving the problem' of defining the identity of objects by stipulating that a data object 'itself' is the information stored in some particular set of memory locations in the computer. This suffices for simple JavaScript programs, but is hardly a general way to resolve the issue of 'sameness' in computational models.

infinite loop. Exercise 3.13 constructed such lists.

Exercise 3.19

Redo exercise 3.18 using an algorithm that takes only a constant amount of space. (This requires a very clever idea.)

Mutation is just assignment

When we introduced compound data, we observed in section **??** that pairs can be represented purely in terms of functions:

```
function pair(x, y) {
    function dispatch(m) {
        if (m === "head") {
            return x;
        } else if (m === "tail") {
            return y;
        } else {
            return "undefined operation -- pair";
        }
    }
    return dispatch;
}
```

The same observation is true for mutable data. We can implement mutable data objects as functions using assignment and local state. For instance, we can extend the above pair implementation to handle `set_head` and `set_tail` in a manner analogous to the way we implemented bank accounts using `make-account` in section 3.1.1:

```
function pair(x, y) {
    function set_x(v) {
        x = v;
    }
    function set_y(v) {
        y = v;
    }
    return function dispatch(m) {
               if (m === "head") {
                   return x;
               } else if (m === "tail") {
                   return y;
               } else if (m === "set_head") {
                   return set_x;
               } else if (m === "set_tail") {
                   return set_y;
```

```
                } else {
                    return "undefined operation - - pair";
                }
            };
}

function head(z) {
    return z("head");
}

function tail(z) {
    return z("tail");
}

function set_head(z, new_value) {
    (z("set_head"))(new_value);
    return z;
}

function set_tail(z, new_value) {
    (z("set_tail"))(new_value);
    return z;
}
```

Assignment is all that is needed, theoretically, to account for the behavior of mutable data. As soon as we admit assignment to our language, we raise all the issues, not only of assignment, but of mutable data in general.[19]

Exercise 3.20

Draw environment diagrams to illustrate the evaluation of the sequence of expressions

```
const x = pair(1, 2);
const z = pair(x, x);
set_head(tail(z), 17);
head(x);
```

using the functional implementation of pairs given above. (Compare exercise 3.11.)

[19] On the other hand, from the viewpoint of implementation, assignment requires us to modify the environment, which is itself a mutable data structure. Thus, assignment and mutation are equipotent: Each can be implemented in terms of the other.

3.3.2 Representing Queues

The mutators `set_head` and `set_tail` enable us to use pairs to construct data structures that cannot be built with `pair`, `head`, and `tail` alone. This section shows how to use pairs to represent a data structure called a queue. Section 3.3.3 will show how to represent data structures called tables.

A *queue* is a sequence in which items are inserted at one end (called the *rear* of the queue) and deleted from the other end (the *front*). Figure 3.18 shows an initially empty queue in which the items `a` and `b` are inserted. Then `a` is removed, `c` and `d` are inserted, and `b` is removed. Because items are always removed in the order in which they are inserted, a queue is sometimes called a *FIFO* (first in, first out) buffer.

Operation	Resulting Queue
`var q = make_queue();`	
`insert_queue(q, "a");`	`a`
`insert_queue(q, "b");`	`a b`
`delete_queue(q);`	`b`
`insert_queue(q, "c");`	`b c`
`insert_queue(q, "d");`	`b c d`
`delete_queue(q);`	`c d`

Figure 3.18: Queue operations.

In terms of data abstraction, we can regard a queue as defined by the following set of operations:

- a constructor:

  `make_queue()` returns an empty queue (a queue containing no items).

- two selectors:

  `empty_queue(queue)` tests if the queue is empty.

  `front_queue(queue)` returns the object at the front of the queue, signaling an error if the queue is empty; it does not modify the queue.

- two mutators:

  `insert_queue(queue, item)` inserts the item at the rear of the queue and returns the modified queue as its value.

  `delete_queue(queue)` removes the item at the front of the queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

Because a queue is a sequence of items, we could certainly represent it as an ordinary list; the front of the queue would be the `head` of the list, inserting an item in the queue would amount to appending a new element at the end of the list, and deleting an item from the queue would just be taking the `tail` of the list. However, this representation is inefficient, because in order to insert an item we must scan the list until we reach the end. Since the only method we have for scanning a list is by successive `tail` operations, this scanning requires $\Theta(n)$ steps for a list of n items. A simple modification to the list representation overcomes this disadvantage by allowing the queue operations to be implemented so that they require $\Theta(1)$ steps; that is, so that the number of steps needed is independent of the length of the queue.

The difficulty with the list representation arises from the need to scan to find the end of the list. The reason we need to scan is that, although the standard way of representing a list as a chain of pairs readily provides us with a pointer to the beginning of the list, it gives us no easily accessible pointer to the end. The modification that avoids the drawback is to represent the queue as a list, together with an additional pointer that indicates the final pair in the list. That way, when we go to insert an item, we can consult the rear pointer and so avoid scanning the list.

A queue is represented, then, as a pair of pointers, `front_ptr` and `rear_ptr`, which indicate, respectively, the first and last pairs in an ordinary list. Since we would like the queue to be an identifiable object, we can use `pair` to combine the two pointers. Thus, the queue itself will be the `pair` of the two pointers. Figure 3.19 illustrates this representation.

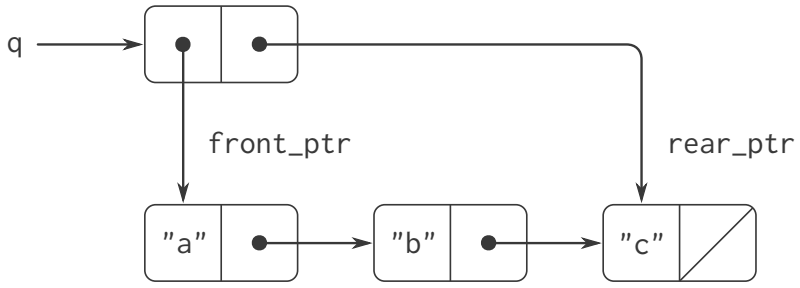


Figure 3.19: Implementation of a queue as a list with front and rear pointers.

To define the queue operations we use the following functions, which enable us to select and to modify the front and rear pointers of a queue:

```
function front_ptr(queue) {
    return head(queue);
}
function rear_ptr(queue) {
    return tail(queue);
}
function set_front_ptr(queue, item) {
    set_head(queue, item);
```

```
}
function set_rear_ptr(queue, item) {
    set_tail(queue, item);
}
```

Now we can implement the actual queue operations. We will consider a queue to be empty if its front pointer is the empty list:

```
function is_empty_queue(queue) {
    return is_null(front_ptr(queue));
}
```

The `make_queue` constructor returns, as an initially empty queue, a pair whose `head` and `tail` are both the empty list:

```
function make_queue() {
    return pair(null, null);
}
```

To select the item at the front of the queue, we return the `head` of the pair indicated by the front pointer:

```
function front_queue(queue) {
    return is_empty_queue(queue)
        ? error(queue, "front_queue called with an empty queue:")
        : front_ptr(queue);
}
```

To insert an item in a queue, we follow the method whose result is indicated in Figure 3.20. We first create a new pair whose `head` is the item to be inserted and whose `tail` is the empty list. If the queue was initially empty, we set the front and rear pointers of the queue to this new pair. Otherwise, we modify the final pair in the queue to point to the new pair, and also set the rear pointer to the new pair.

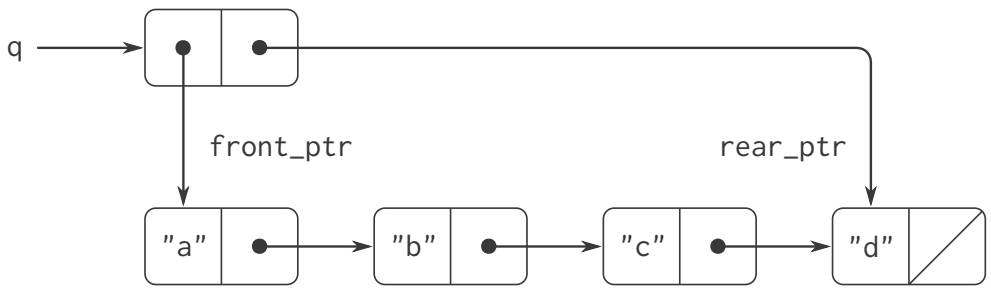


Figure 3.20: Result of using `insert_queue(q, 'd')` on the queue of Figure 3.19.

```
function insert_queue(queue, item) {
    const new_pair = pair(item, null);
    if (is_empty_queue(queue)) {
        set_front_ptr(queue, new_pair);
        set_rear_ptr(queue, new_pair);
    } else {
        set_tail(rear_ptr(queue), new_pair);
        set_rear_ptr(queue, new_pair);
    }
    return queue;
}
```

To delete the item at the front of the queue, we merely modify the front pointer so that it now points at the second item in the queue, which can be found by following the `tail` pointer of the first item (see Figure 3.21):[20]

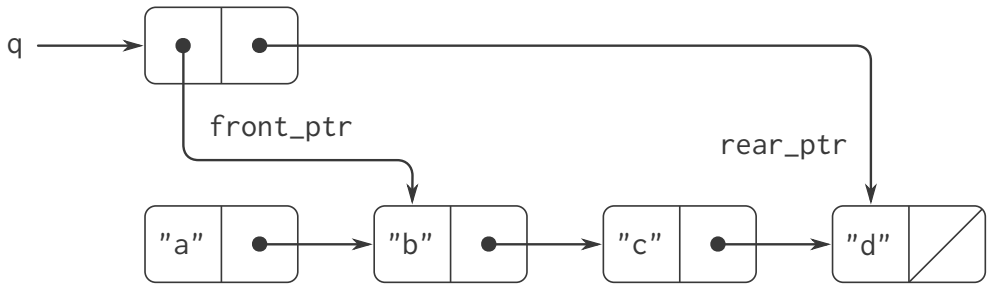


Figure 3.21: Result of using `delete_queue(q)` on the queue of Figure 3.20.

```
function delete_queue(queue) {
    if (is_empty_queue(queue)) {
        error(queue, "delete_queue called with an empty queue:");
    } else {
        set_front_ptr(queue, tail(front_ptr(queue)));
        return queue;
    }
}
```

Exercise 3.21

Ben Bitdiddle decides to test the queue implementation described above. He types in the functions to the JavaScript interpreter and proceeds to try them out:

[20] If the first item is the final item in the queue, the front pointer will be the empty list after the deletion, which will mark the queue as empty; we needn't worry about updating the rear pointer, which will still point to the deleted item, because `empty-queue?` looks only at the front pointer.

```
const q1 = make_queue();
insert_queue(q1, "a");
insert_queue(q1, "b");
delete_queue(q1);
delete_queue(q1);
```

'It's all wrong!' he complains. 'The interpreter's response shows that the last item is inserted into the queue twice. And when I delete both items, the second `b` is still there, so the queue isn't empty, even though it's supposed to be.' Eva Lu Ator suggests that Ben has misunderstood what is happening. 'It's not that the items are going into the queue twice,' she explains. 'It's just that the standard JavaScript printer doesn't know how to make sense of the queue representation. If you want to see the queue printed correctly, you'll have to define your own print function for queues.' Explain what Eva Lu is talking about. In particular, show why Ben's examples produce the printed results that they do. Define a function `print_queue` that takes a queue as input and prints the sequence of items in the queue. Solution

Exercise 3.22

Instead of representing a queue as a pair of pointers, we can build a queue as a function with local state. The local state will consist of pointers to the beginning and the end of an ordinary list. Thus, the `make_queue` function will have the form

```
function make_queue() {
    function front_ptr(...) ...
    function rear_ptr(...) ...
    //definitions of internal functions
    function dispatch(m) ...
    return dispatch;
}
```

Complete the definition of `make_queue` and provide implementations of the queue operations using this representation.

Exercise 3.23

A *deque* ('double-ended queue') is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make_deque`, the predicate `is_empty_deque`, selectors `front_deque` `front-deque` and `rear_deque` and mutators `front_insert_deque`, `front_delete_deque`, `rear_insert_deque`, and `rear_delete_deque`.

Show how to represent deques using pairs, and give implementations of the operations.[21] All operations should be accomplished in $\Theta(1)$ steps.

3.3.3 Representing Tables

When we studied various ways of representing sets in chapter 2, we mentioned in section ?? the task of maintaining a table of records indexed by identifying keys. In the implementation of data-directed programming in section 2.2.3, we made extensive use of two-dimensional tables, in which information is stored and retrieved using two keys. Here we see how to build tables as mutable list structures.

We first consider a one-dimensional table, in which each value is stored under a single key. We implement the table as a list of records, each of which is implemented as a pair consisting of a key and the associated value. The records are glued together to form a list by pairs whose heads point to successive records. These gluing pairs are called the *backbone* of the table. In order to have a place that we can change when we add a new record to the table, we build the table as a *headed list*. A headed list has a special backbone pair at the beginning, which holds a dummy 'record'—in this case the arbitrarily chosen string `"*table*"`. Figure 3.22 shows the box-and-pointer diagram for the table

```
a: 1
b: 2
c: 3
```

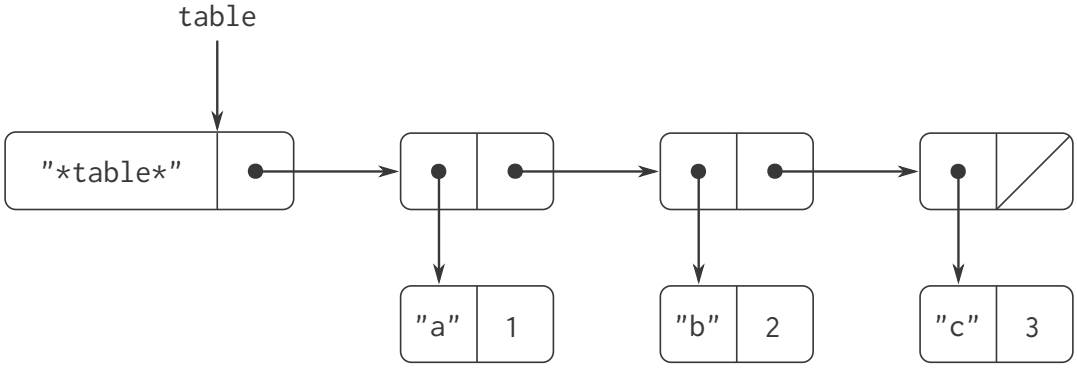


Figure 3.22: A table represented as a headed list.

To extract information from a table we use the `lookup` function, which takes a key as argument and returns the associated value (or false if there is no value stored under that key). The function `lookup` is defined in terms of the `assoc` operation, which expects a key and a list of records as arguments. Note that `assoc` never sees the dummy record. The function `assoc` returns the

[21] Be careful not to make the interpreter try to print a structure that contains cycles. (See exercise 3.13.)

record that has the given key as its head.[22] The function lookup then checks to see that the resulting record returned by assoc is not false, and returns the value (the tail) of the record.

```
function lookup(key, table) {
    const record = assoc(key, tail(table));
    return record === undefined
           ? undefined
           : tail(record);
}
```

```
function assoc(key, records) {
    return is_null(records)
           ? undefined
           : is_equal(key, head(head(records)))
             ? head(records)
             : assoc(key, tail(records));
}
```

To insert a value in a table under a specified key, we first use assoc to see if there is already a record in the table with this key. If not, we form a new record by pairing the key with the value, and insert this at the head of the table's list of records, after the dummy record. If there already is a record with this key, we set the tail of this record to the designated new value. The header of the table provides us with a fixed location to modify in order to insert the new record.[23]

```
function insert(key, value, table) {
    const record = assoc(key, tail(table));
    return record === undefined
           ? set_tail(table, pair(pair(key, value),
                                  tail(table)))
           : set_tail(record, value);
}
```

To construct a new table, we simply create a list containing the symbol *table*:

```
function make_table() {
    return list("*table*");
}
```

[22] Because assoc uses is_equal, it can recognize keys that are strings, numbers, or list structure.

[23] Thus, the first backbone pair is the object that represents the table 'itself'; that is, a pointer to the table is a pointer to this pair. This same backbone pair always starts the table. If we did not arrange things in this way, insert would have to return a new value for the start of the table when it added a new record.

Two-dimensional tables

In a two-dimensional table, each value is indexed by two keys. We can construct such a table as a one-dimensional table in which each key identifies a subtable. Figure 3.23 shows the box-and-pointer diagram for the table

```
math:
    +:  43
    -:  45
    *:  42
letters:
    a:  97
    b:  98
```

which has two subtables. (The subtables don't need a special header symbol, since the key that identifies the subtable serves this purpose.)

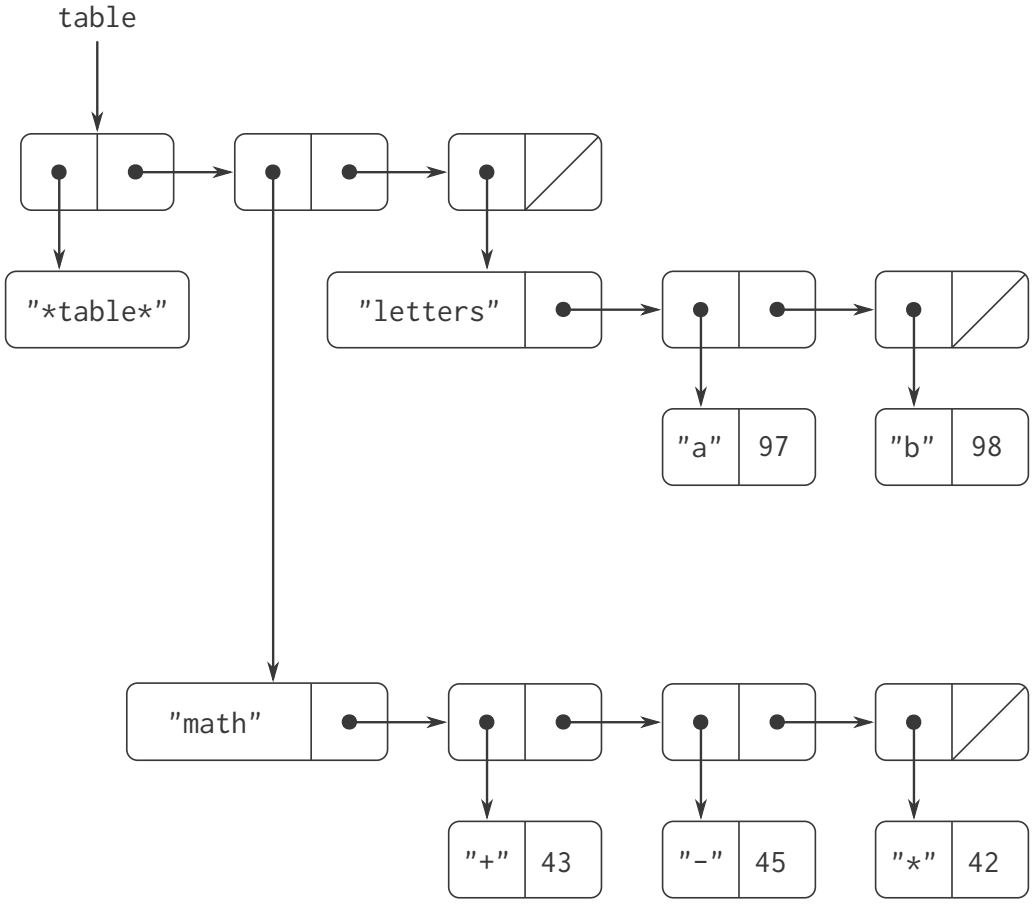


Figure 3.23: A two-dimensional table.

When we look up an item, we use the first key to identify the correct subtable. Then we use the second key to identify the record within the subtable.

```
function lookup(key_1, key_2, table) {
    const subtable = assoc(key_1, tail(table));
    if (subtable === undefined) {
        return undefined;
    } else {
        const record = assoc(key_2, tail(subtable));
        if (record === undefined) {
            return undefined;
        } else {
            return tail(record);
        }
    }
}
```

To insert a new item under a pair of keys, we use assoc to see if there is a subtable stored under the first key. If not, we build a new subtable containing the single record (key_2, value) and insert it into the table under the first key. If a subtable already exists for the first key, we insert the new record into this subtable, using the insertion method for one-dimensional tables described above:

```
function insert(key_1, key_2, value, table) {
    const subtable = assoc(key_1, tail(table));
    if (subtable === undefined) {
        set_tail(table,
                 pair(list(key_1, pair(key_2, value)),
                      tail(table)));
    } else {
        const record = assoc(key_2, tail(table));
        if (record === undefined) {
            set_tail(subtable,
                     pair(pair(key_2, value),
                          tail(subtable)));
        } else {
            set_tail(record, value);
        }
    }
}
```

Creating local tables

The `lookup` and `insert` operations defined above take the table as an argument. This enables us to use programs that access more than one table. Another way to deal with multiple tables is to have separate `lookup` and `insert` functions for each table. We can do this by representing a table procedurally, as an object that maintains an internal table as part of its local state. When sent an appropriate message, this 'table object' supplies the function with which to operate on the internal table. Here is a generator for two-dimensional tables represented in this fashion:

```
function make_table() {
    const local_table = list("*table*");
    function lookup(key_1, key_2) {
        const subtable = assoc(key_1, tail(local_table));
        if (subtable === undefined) {
            return undefined;
        } else {
            const record = assoc(key_2, tail(subtable));
            if (record === undefined) {
                return undefined;
            } else {
                return tail(record);
            }
        }
    }
    function insert(key_1, key_2, value) {
        const subtable = assoc(key_1, tail(local_table));
        if (subtable === undefined) {
            set_tail(local_table,
                     pair(list(key_1, pair(key_2, value)),
                          tail(local_table)));
        } else {
            const record = assoc(key_2, tail(subtable));
            if (record === undefined) {
                set_tail(subtable,
                                pair(pair(key_2, value),
                               tail(subtable)));
            } else {
                set_tail(record, value);
            }
        }
    }
    function dispatch(m) {
        return m === "lookup"
               ? lookup
               : m === "insert"
                 ? insert
                 : "undefined operation -- table";
```

```
    }
    return dispatch;
}
```

Using `make_table`, we could implement the `get` and `put` operations used in section 2.2.3 for data-directed programming, as follows:

```
const operation_table = make_table();
const get = operation_table("lookup");
const put = operation_table("insert");
```

The function `get` takes as arguments two keys, and `put` takes as arguments two keys and a value. Both operations access the same local table, which is encapsulated within the object created by the call to `make_table`.

Exercise 3.24

In the table implementations above, the keys are tested for equality using `is_equal` (called by `assoc`). This is not always the appropriate test. For instance, we might have a table with numeric keys in which we don't need an exact match to the number we're looking up, but only a number within some tolerance of it. Design a table constructor `make_table` that takes as an argument a `same_key` function that will be used to test 'equality' of keys. The function `make_table` should return a `dispatch` function that can be used to access appropriate `lookup` and `insert` functions for a local table.

Exercise 3.25

Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The `lookup` and `insert` functions should take as input a list of keys used to access the table.

Exercise 3.26

To search a table as implemented above, one needs to scan through the list of records. This is basically the unordered list representation of section **??**. For large tables, it may be more efficient to structure the table in a different manner. Describe a table implementation where the (key, value) records are organized using a binary tree, assuming that keys can be ordered in some way (e.g., numerically or alphabetically). (Compare exercise **??** of chapter 2.)

Exercise 3.27

Memoization (also called *tabulation*) is a technique that enables a function to record, in a local table, values that have previously been computed. This technique can make a vast difference in the performance of a program. A memoized function maintains a table in which values of previous calls are stored using as keys the arguments that produced the values. When the memoized function is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table. As an example of memoization, recall from section **??** the exponential process for computing Fibonacci numbers:

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

The memoized version of the same function is

```
const memo_fib = memoize(n => n === 0
                              ? 0
                              : n === 1
                                ? 1
                                : memo_fib(n - 1) +
                                  memo_fib(n - 2)
                         );
```

where the memoizer is defined as

```
function memoize(f) {
    const table = make_table();
    return x => {
        const previously_computed_result
            = lookup(x, table);
        if (previously_computed_result === undefined) {
            const result = f(x);
            insert(x, result, table);
            return result;
        } else {
            return previously_computed_result;
        }
    };
}
```

Draw an environment diagram to analyze the computation of `memo_fib(3)`. Explain why `memo_fib` computes the nth Fibonacci number in a number of steps proportional to n. Would the scheme still work if we had simply defined `memo_fib` to be `memoize(fib)`?

3.3.4 A Simulator for Digital Circuits

Designing complex digital systems, such as computers, is an important engineering activity. Digital systems are constructed by interconnecting simple elements. Although the behavior of these individual elements is simple, networks of them can have very complex behavior. Computer simulation of proposed circuit designs is an important tool used by digital systems engineers. In this section we design a system for performing digital logic simulations. This system typifies a kind of program called an *event-driven simulation*, in which actions ('events') trigger further events that happen at a later time, which in turn trigger more events, and so so.

Our computational model of a circuit will be composed of objects that correspond to the elementary components from which the circuit is constructed. There are *wires*, which carry *digital signals*. A digital signal may at any moment have only one of two possible values, 0 and 1. There are also various types of digital *function boxes*, which connect wires carrying input signals to other output wires. Such boxes produce output signals computed from their input signals. The output signal is delayed by a time that depends on the type of the function box. For example, an *inverter* is a primitive function box that inverts its input. If the input signal to an inverter changes to 0, then one *inverter-delay* later the inverter will change its output signal to 1. If the input signal to an inverter changes to 1, then one *inverter-delay* later the inverter will change its output signal to 0. We draw an inverter symbolically as in Figure 3.24. An *and-gate*, also shown in Figure 3.24, is a primitive function box with two inputs and one output. It drives its output signal to a value that is the *logical and* of the inputs. That is, if both of its input signals become 1, then one *and-gate-delay* time later the and-gate will force its output signal to be 1; otherwise the output will be 0. An *or-gate* is a similar two-input primitive function box that drives its output signal to a value that is the *logical or* of the inputs. That is, the output will become 1 if at least one of the input signals is 1; otherwise the output will become 0.

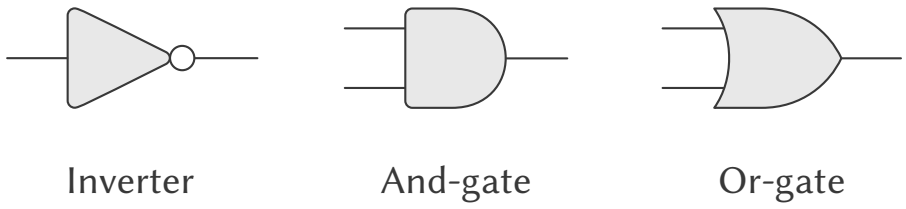


Figure 3.24: Primitive functions in the digital logic simulator.

We can connect primitive functions together to construct more complex functions. To accomplish this we wire the outputs of some function boxes to the inputs of other function boxes. For example, the *half-adder* circuit shown in Figure 3.25 consists of an or-gate, two and-gates, and an inverter. It takes two input signals, A and B, and has two output signals, S and C. S will become 1 whenever precisely one of A and B is 1, and C will become 1 whenever A and B are

both 1. We can see from the figure that, because of the delays involved, the outputs may be generated at different times. Many of the difficulties in the design of digital circuits arise from this fact.

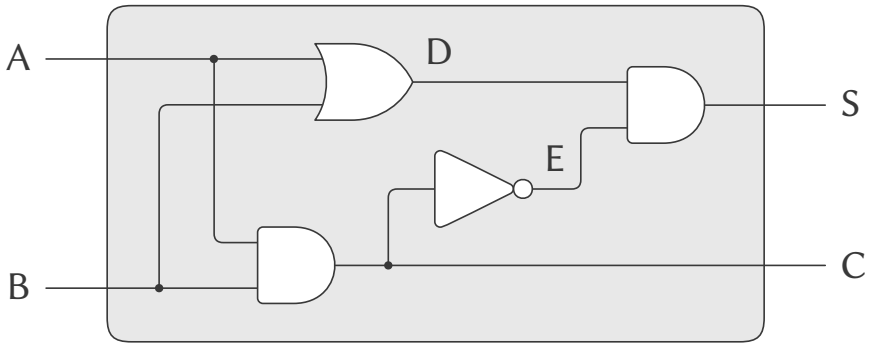


Figure 3.25: A half-adder circuit.

We will now build a program for modeling the digital logic circuits we wish to study. The program will construct computational objects modeling the wires, which will 'hold' the signals. Function boxes will be modeled by functions that enforce the correct relationships among the signals.

One basic element of our simulation will be a function `make_wire`, which constructs wires. For example, we can construct six wires as follows:

```
const a = make_wire();
const b = make_wire();
const c = make_wire();
const d = make_wire();
const e = make_wire();
const s = make_wire();
```

We attach a function box to a set of wires by calling a function that constructs that kind of box. The arguments to the constructor function are the wires to be attached to the box. For example, given that we can construct and-gates, or-gates, and inverters, we can wire together the half-adder shown in Figure 3.25:

```
or_gate(a, b, d);

and_gate(a, b, c);

inverter(c, e);

and_gate(d, e, s);
```

Better yet, we can explicitly name this operation by defining a function `half_adder` that constructs this circuit, given the four external wires to be attached to the half-adder:

```
function half_adder(a, b, s, c) {
    const d = make_wire();
    const e = make_wire();
    or_gate(a, b, d);
    and_gate(a, b, c);
    inverter(c, e);
    and_gate(d, e, s);
    return "ok";
}
```

The advantage of making this definition is that we can use `half_adder` itself as a building block in creating more complex circuits. Figure 3.26, for example, shows a *full-adder* composed of two half-adders and an or-gate.[24] We can construct a full-adder as follows:

```
function full_adder(a, b, c_in, sum, c_out) {
    const s = make_wire();
    const c1 = make_wire();
    const c2 = make_wire();
    half_adder(b, c_in, s, c1);
    half_adder(a, s, sum, c2);
    or_gate(c1, c2, c_out);
    return "ok";
}
```

Having defined `full_adder` as a function, we can now use it as a building block for creating still more complex circuits. (For example, see exercise 3.30.)

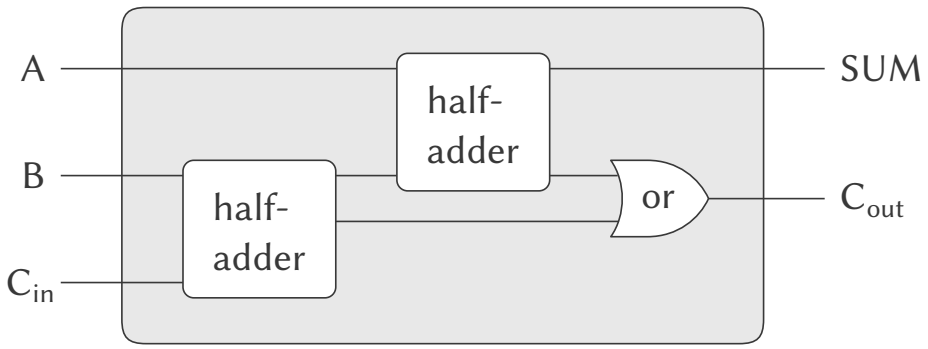


Figure 3.26: A full-adder circuit.

In essence, our simulator provides us with the tools to construct a language of circuits. If we adopt the general perspective on languages with which we approached the study of JavaScript in section **??**, we can say that the primitive function boxes form the primitive elements of the

[24] A full-adder is a basic circuit element used in adding two binary numbers. Here A and B are the bits at corresponding positions in the two numbers to be added, and C_{in} is the carry bit from the addition one place to the right. The circuit generates SUM, which is the sum bit in the corresponding position, and C_{out}, which is the carry bit to be propagated to the left.

language, that wiring boxes together provides a means of combination, and that specifying wiring patterns as functions serves as a means of abstraction.

Primitive function boxes

The primitive function boxes implement the 'forces' by which a change in the signal on one wire influences the signals on other wires. To build function boxes, we use the following operations on wires:

- `get_signal(wire)`: returns the current value of the signal on the wire.
- `set_signal(wire, new_value)`: changes the value of the signal on the wire to the new value.
- `add_action(wire, nullary_function)`: asserts that the designated function should be run whenever the signal on the wire changes value. Such functions are the vehicles by which changes in the signal value on the wire are communicated to other wires.

In addition, we will make use of a function `after_delay` that takes a time delay and a function to be run and executes the given function after the given delay.

Using these functions, we can define the primitive digital logic functions. To connect an input to an output through an inverter, we use `add_action` to associate with the input wire a function that will be run whenever the signal on the input wire changes value. The function computes the `logical_not` of the input signal, and then, after one `inverter_delay`, sets the output signal to be this new value:

```
function inverter(input, output) {
    function invert_input() {
        const new_value = logical_not(get_signal(input));
        after_delay(inverter_delay,
                    () => set_signal(output, new_value));
    }
    add_action(input, invert_input);
    return "ok";
}

function logical_not(s) {
    return s === 0
           ? 1
           : s === 1
             ? 0
             : Error("Invalid signal for logical_not", s);
}
```

An and-gate is a little more complex. The action function must be run if either of the inputs to the gate changes. It computes the `logical_and` (using a function analogous to `logical_not`) of the values of the signals on the input wires and sets up a change to the new value to occur on the output wire after one `and_gate_delay`.

```
function and_gate(a1, a2, output) {
    function and_action_function() {
        const new_value = logical_and(get_signal(a1),
                                      get_signal(a2));
        after_delay(and_gate_delay,
                    () => set_signal(output, new_value);
    }
    add_action(a1, and_action_function);
    add_action(a2, and_action_function);
    return "ok";
}
```

Exercise 3.28

Define an or-gate as a primitive function box. Your `or_gate` constructor should be similar to `and_gate`.

Exercise 3.29

Another way to construct an or-gate is as a compound digital logic device, built from and-gates and inverters. Define a function `or_gate` that accomplishes this. What is the delay time of the or-gate in terms of `and_gate_delay` and `inverter_delay`?

Exercise 3.30

Figure 3.27 shows a *ripple-carry adder* formed by stringing together n full-adders. This is the simplest form of parallel adder for adding two n-bit binary numbers. The inputs $A_1, A_2, A_3, \ldots, A_n$ and $B_1, B_2, B_3, \ldots, B_n$ are the two binary numbers to be added (each A_k and B_k is a 0 or a 1). The circuit generates $S_1, S_2, S_3, \ldots, S_n$, the n bits of the sum, and C, the carry from the addition. Write a function `ripple_carry_adder` that generates this circuit. The function should take as arguments three lists of n wires each—the A_k, the B_k, and the S_k—and also another wire C. The major drawback of the ripple-carry adder is the need to wait for the carry signals to propagate. What is the delay needed to obtain the complete output from an n-bit ripple-carry adder, expressed in terms of the delays for and-gates, or-gates, and inverters?

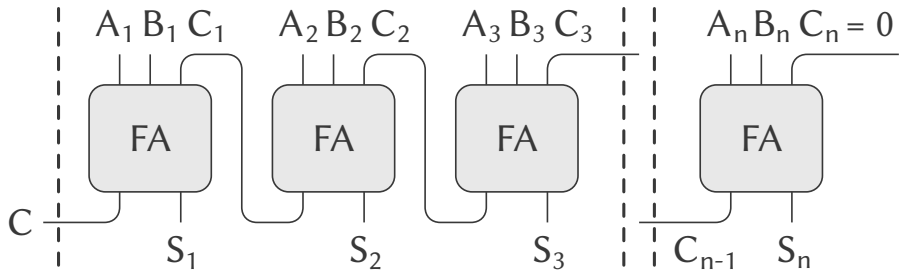


Figure 3.27: A ripple-carry adder for n-bit numbers.

Representing wires

A wire in our simulation will be a computational object with two local state variables: a `signal_value` (initially taken to be 0) and a collection of `action_function` to be run when the signal changes value. We implement the wire, using message-passing style, as a collection of local functions together with a `dispatch` function that selects the appropriate local operation, just as we did with the simple bank-account object in section 3.1.1:

```
function make_wire() {
    let signal_value = 0;
    let action_functions = null;
    function set_my_signal(new_value) {
        if (signal_value !== new_value) {
            signal_value = new_value;
            call_each(action_functions);
        } else {}
    }
    function accept_action_function(fun) {
        action_functions = pair(fun, action_functions);
        fun();
    }
    function dispatch(m) {
        return m === "get_signal"
               ? signal_value
               : m === "set_signal"
                 ? set_my_signal
                 : m === "add_action"
                   ? accept_action_function
                   : Error("Unknown operation in wire", m);
    }
    return dispatch;
}
```

The local function `set_my_signal` tests whether the new signal value changes the signal on the wire. If so, it runs each of the action functions, using the following function `call_each`,

which calls each of the items in a list of no-argument functions:

```
function call_each(functions) {
    if (is_null(functions)) {
        return "done";
    } else {
        (head(functions))();
        call_each(tail(functions));
    }
}
```

The local function `accept_action_function` adds the given function to the list of functions to be run, and then runs the new function once. (See exercise 3.31.)

With the local `dispatch` function set up as specified, we can provide the following functions to access the local operations on wires:[25]

```
function get_signal(wire) {
    return wire("get_signal");
}

function set_signal(wire, new_value) {
    return (wire("set_signal"))(new_value);
}

function add_action(wire, action_function) {
    return (wire("add_action"))(action_function);
}
```

Wires, which have time-varying signals and may be incrementally attached to devices, are typical of mutable objects. We have modeled them as functions with local state variables that are modified by assignment. When a new wire is created, a new set of state variables is allocated (by the **`let`** expression in `make_wire`) and a new `dispatch` function is constructed and returned, capturing the environment with the new state variables.

The wires are shared among the various devices that have been connected to them. Thus, a change made by an interaction with one device will affect all the other devices attached to the wire. The wire communicates the change to its neighbors by calling the action functions provided to it when the connections were established.

[25] These functions are simply syntactic sugar that allow us to use ordinary functional syntax to access the local functions of objects. It is striking that we can interchange the role of 'functions' and 'data' in such a simple way. For example, if we write `wire('get_signal')` we think of `wire` as a function that is called with the message `"get_signal"` as input. Alternatively, writing `get_signal(wire)` encourages us to think of `wire` as a data object that is the input to a function `get_signal`. The truth of the matter is that, in a language in which we can deal with functions as objects, there is no fundamental difference between ' functions ' and 'data,' and we can choose our syntactic sugar to allow us to program in whatever style we choose.

The agenda

The only thing needed to complete the simulator is `after_delay`. The idea here is that we maintain a data structure, called an *agenda*, that contains a schedule of things to do. The following operations are defined for agendas:

- `make_agenda()`: returns a new empty agenda.
- `is_empty_agenda(agenda)`: is true if the specified agenda is empty.
- `first_agenda_item(agenda)`: returns the first item on the agenda.
- `remove_first_agenda_item(agenda)`: modifies the agenda by removing the first item.
- `add_to_agenda(time, action, agenda)`: modifies the agenda by adding the given action function to be run at the specified time.
- `current_time(agenda)`:: returns the current simulation time.

The particular agenda that we use is denoted by `the_agenda` . The function `after_delay` adds new elements to `the_agenda` :

```
function after_delay(delay, action) {
    add_to_agenda(delay + current_time(the_agenda),
                  action, the_agenda);
}
```

The simulation is driven by the function `propagate`, which operates on `the_agenda` , executing each function on the agenda in sequence. In general, as the simulation runs, new items will be added to the agenda, and `propagate` will continue the simulation as long as there are items on the agenda:

```
function propagate() {
    if (is_empty_agenda(the_agenda)) {
        return "done";
    } else {
        const first = first_agenda_item(the_agenda);
        first();
        remove_first_agenda_item(the_agenda);
        return propagate();
    }
}
```

A sample simulation

The following function, which places a 'probe' on a wire, shows the simulator in action. The probe tells the wire that, whenever its signal changes value, it should print the new signal value, together with the current time and a name that identifies the wire:

```
function probe(name, wire) {
    add_action(wire,
               () => display(name + " " +
                             current_time(the_agenda) +
                             ", New value = " +
                             get_signal(wire));
}
```

We begin by initializing the agenda and specifying delays for the primitive function boxes:

```
const the_agenda = make_agenda();
const inverter_delay = 2;
const and_gate_delay = 3;
const or_gate_delay = 5;
```

Now we define four wires, placing probes on two of them:

```
const input_1 = make_wire();
const input_2 = make_wire();
const sum = make_wire();
const carry = make_wire();

probe("Sum", sum);
// Sum 0, New value = 0

probe("Carry", carry);
// Carry 0, New value = 0
```

Next we connect the wires in a half-adder circuit (as in Figure 3.25), set the signal on `input_1` to 1, and run the simulation:

```
half_adder(input_1, input_2, sum, carry);

set_signal(input_1, 1);

propagate();
// Sum 8, New Value = 1
```

The `sum` signal changes to 1 at time 8. We are now eight time units from the beginning of the simulation. At this point, we can set the signal on `input_2` to 1 and allow the values to propagate:

```
set_signal(input_2, 1);
```

```
propagate();
// Carry 11, New value = 1
// Sum 16, New value = 0
```

The carry changes to 1 at time 11 and the sum changes to 0 at time 16.

Exercise 3.31

The internal function accept_action_function defined in make_wire specifies that when a new action function is added to a wire, the function is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the system's response would differ if we had defined accept_action_function as

```
function accept_action_function(fun) {
    action_functions = pair(fun, action_functions);
}
```

Implementing the agenda

Finally, we give details of the agenda data structure, which holds the functions that are scheduled for future execution.

The agenda is made up of *time segments*. Each time segment is a pair consisting of a number (the time) and a queue (see exercise 3.32) that holds the functions that are scheduled to be run during that time segment.

```
function make_time_segment(time, queue) {
    return pair(time, queue);
}

function segment_time(s) {
    return head(s);
}

function segment_queue(s) {
    return tail(s);
}
```

We will operate on the time-segment queues using the queue operations described in section 3.3.2.

The agenda itself is a one-dimensional table of time segments. It differs from the tables described in section 3.3.3 in that the segments will be sorted in order of increasing time. In addition, we store the *current time* (i.e., the time of the last action that was processed) at the

head of the agenda. A newly constructed agenda has no time segments and has a current time of 0:[26]

```
function make_agenda() {
    return list(0);
}

function current_time(agenda) {
    return head(agenda);
}

function set_current_time(agenda, time) {
    set_head(agenda, time);
}

function segments(agenda) {
    return tail(agenda);
}

function set_segments(agenda, segs) {
    set_tail(agenda, segs);
}

function first_segment(agenda) {
    return head(segments(agenda));
}

function rest_segments(agenda) {
    return tail(segments(agenda));
}
```

An agenda is empty if it has no time segments:

```
function is_empty_agenda(agenda) {
    return is_null(segments(agenda));
}
```

To add an action to an agenda, we first check if the agenda is empty. If so, we create a time segment for the action and install this in the agenda. Otherwise, we scan the agenda, examining the time of each segment. If we find a segment for our appointed time, we add the action to the associated queue. If we reach a time later than the one to which we are appointed, we insert a new time segment into the agenda just before it. If we reach the end of the agenda, we must create a new time segment at the end.

```
function add_to_agenda(time, action, agenda) {
    function belongs_before(segs) {
```

[26] The agenda is a headed list, like the tables in section 3.3.3, but since the list is headed by the time, we do not need an additional dummy header (such as the `*table*` symbol used with tables).

```
        return is_null(segs) ||
               time < segment_time(head(segs));
    }
    function make_new_time_segment(time, action) {
        const q = make_queue();
        insert_queue(q, action);
        return make_time_segment(time, q);
    }
    function add_to_segments(segs) {
        if (segment_time(head(segs)) === time) {
            insert_queue(segment_queue(head(segs)), action);
        } else {
            const rest = tail(segs);
            if (belongs_before(rest)) {
                set_tail(segs,
                    pair(make_new_time_segment(time, action),
                         tail(segs)));
            } else {
                add_to_segments(rest);
            }
        }
    }
    const segs = segments(agenda);
    if (belongs_before(segs)) {
        set_segments(agenda,
            pair(make_new_time_segment(time, action),
                 segs));
    } else {
        add_to_segments(segs);
    }
}
```

The function that removes the first item from the agenda deletes the item at the front of the queue in the first time segment. If this deletion makes the time segment empty, we remove it from the list of segments:[27]

```
function remove_first_agenda_item(agenda) {
    const q = segment_queue(first_segment(agenda));
    delete_queue(q);
    if (is_empty_queue(q)) {
        set_segments(agenda, rest_segments(agenda));
    } else {}
}
```

The first agenda item is found at the head of the queue in the first time segment. Whenever

[27] Observe that the **if** expression in this function has no alternative expression. Such a 'one-armed **if** statement' is used to decide whether to do something, rather than to select between two expressions. An **if** expression returns an unspecified value if the predicate is false and there is no alternative.

we extract an item, we also update the current time:[28]

```
function first_agenda_item(agenda) {
    if (is_empty_agenda(agenda)) {
        error("Agenda is empty: first_agenda_item");
    } else {
        const first_seg = first_segment(agenda);
        set_current_time(agenda, segment_time(first_seg));
        return front_queue(segment_queue(first_seg));
    }
}
```

Exercise 3.32

The functions to be run during each time segment of the agenda are kept in a queue. Thus, the functions for each segment are called in the order in which they were added to the agenda (first in, first out). Explain why this order must be used. In particular, trace the behavior of an and-gate whose inputs change from 0,1 to 1,0 in the same segment and say how the behavior would differ if we stored a segment's functions in an ordinary list, adding and removing functions only at the front (last in, first out).

3.3.5 Propagation of Constraints

Computer programs are traditionally organized as one-directional computations, which perform operations on prespecified arguments to produce desired outputs. On the other hand, we often model systems in terms of relations among quantities. For example, a mathematical model of a mechanical structure might include the information that the deflection d of a metal rod is related to the force F on the rod, the length L of the rod, the cross-sectional area A, and the elastic modulus E via the equation

$$dAE = FL$$

Such an equation is not one-directional. Given any four of the quantities, we can use it to compute the fifth. Yet translating the equation into a traditional computer language would force us to choose one of the quantities to be computed in terms of the other four. Thus, a function for computing the area A could not be used to compute the deflection d, even though the computations of A and d arise from the same equation.[29]

[28] In this way, the current time will always be the time of the action most recently processed. Storing this time at the head of the agenda ensures that it will still be available even if the associated time segment has been deleted.

[29] Constraint propagation first appeared in the incredibly forward-looking SKETCHPAD system of Ivan Sutherland (1963). A beautiful constraint-propagation system based on the Smalltalk language was developed by Alan

In this section, we sketch the design of a language that enables us to work in terms of relations themselves. The primitive elements of the language are *primitive constraints*, which state that certain relations hold between quantities. For example, `adder(a, b, c)` specifies that the quantities a, b, and c must be related by the equation $a+b=c$, `multiplier(x, y, z)` expresses the constraint $xy=z$, and `constant(3.14, x)` says that the value of x must be 3.14.

Our language provides a means of combining primitive constraints in order to express more complex relations. We combine constraints by constructing *constraint networks*, in which constraints are joined by *connectors*. A connector is an object that 'holds' a value that may participate in one or more constraints. For example, we know that the relationship between Fahrenheit and Celsius temperatures is

$$9C = 5(F - 32)$$

Such a constraint can be thought of as a network consisting of primitive adder, multiplier, and constant constraints (figure 3.28). In the figure, we see on the left a multiplier box with three terminals, labeled $m1$, $m2$, and p. These connect the multiplier to the rest of the network as follows: The $m1$ terminal is linked to a connector C, which will hold the Celsius temperature. The $m2$ terminal is linked to a connector w, which is also linked to a constant box that holds 9. The p terminal, which the multiplier box constrains to be the product of $m1$ and $m2$, is linked to the p terminal of another multiplier box, whose $m2$ is connected to a constant 5 and whose $m1$ is connected to one of the terms in a sum.

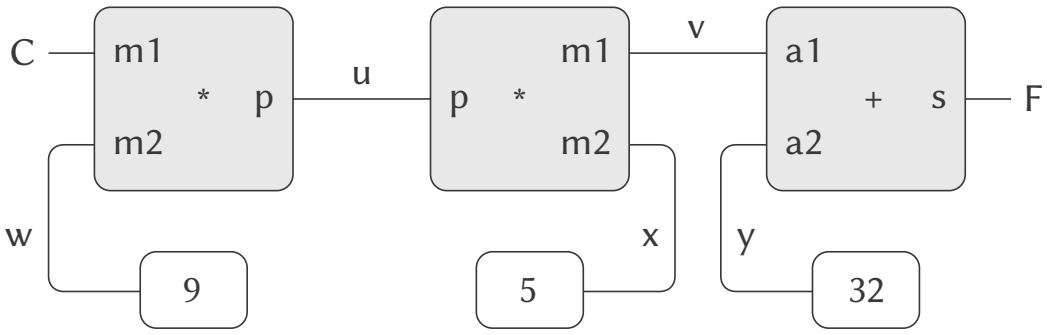


Figure 3.28: The relation $9C = 5(F - 32)$ expressed as a constraint network.

Computation by such a network proceeds as follows: When a connector is given a value (by the user or by a constraint box to which it is linked), it awakens all of its associated constraints (except for the constraint that just awakened it) to inform them that it has a value. Each awakened constraint box then polls its connectors to see if there is enough information to determine a value for a connector. If so, the box sets that connector, which then awakens

Borning (1977) at Xerox Palo Alto Research Center. Sussman, Stallman, and Steele applied constraint propagation to electrical circuit analysis (Sussman and Stallman 1975; Sussman and Steele 1980). TK!Solver (Konopasek and Jayaraman 1984) is an extensive modeling environment based on constraints.

all of its associated constraints, and so on. For instance, in conversion between Celsius and Fahrenheit, w, x, and y are immediately set by the constant boxes to 9, 5, and 32, respectively. The connectors awaken the multipliers and the adder, which determine that there is not enough information to proceed. If the user (or some other part of the network) sets C to a value (say 25), the leftmost multiplier will be awakened, and it will set u to $25 \cdot 9 = 225$. Then u awakens the second multiplier, which sets v to 45, and v awakens the adder, which sets F to 77.

Using the constraint system

To use the constraint system to carry out the temperature computation outlined above, we first create two connectors, `C` and `F`, by calling the constructor `make_connector`, and link `C` and `F` in an appropriate network:

```
const C = make_connector();
const F = make_connector();
celsius_fahrenheit_converter(C, F);
```

The function that creates the network is defined as follows:

```
function celsius_fahrenheit_converter(c, f) {
    const u = make_connector();
    const v = make_connector();
    const w = make_connector();
    const x = make_connector();
    const y = make_connector();
    multiplier(c, w, u);
    multiplier(v, x, u);
    adder(v, y, f);
    constant(9, w);
    constant(5, x);
    constant(32, y);
    return "ok";
}
```

This function creates the internal connectors `u`, `v`, `w`, `x`, and `y`, and links them as shown in Figure 3.28 using the primitive constraint constructors `adder`, `multiplier`, and `constant`. Just as with the digital-circuit simulator of section 3.3.4, expressing these combinations of primitive elements in terms of functions automatically provides our language with a means of abstraction for compound objects.

To watch the network in action, we can place probes on the connectors `C` and `F`, using a `probe` function similar to the one we used to monitor wires in section 3.3.4. Placing a probe on a connector will cause a message to be printed whenever the connector is given a value:

```
probe("Celsius Temp", C);
probe("Fahrenheit Temp", F);
```

Next we set the value of C to 25. (The third argument to set_value tells C that this directive comes from the user.)

```
set_value(C, 25, "user");
// Probe: Celsius Temp = 25
// Probe: Fahrenheit Temp = 77
```

The probe on C awakens and reports the value. C also propagates its value through the network as described above. This sets F to 77, which is reported by the probe on F.

Now we can try to set F to a new value, say 212:

```
set_value(F, 212, "user");
// Error! Contradiction (77 212)
```

The connector complains that it has sensed a contradiction: Its value is 77, and someone is trying to set it to 212. If we really want to reuse the network with new values, we can tell C to forget its old value:

```
forget_value(C, "user");
// Probe: Celsius Temp = ?
// Probe: Fahrenheit Temp = ?
```

C finds that the "user", who set its value originally, is now retracting that value, so C agrees to lose its value, as shown by the probe, and informs the rest of the network of this fact. This information eventually propagates to F, which now finds that it has no reason for continuing to believe that its own value is 77. Thus, F also gives up its value, as shown by the probe.

Now that F has no value, we are free to set it to 212:

```
set_value(F, 212, "user");
// Probe: Fahrenheit Temp = 212
// Probe: Celsius Temp = 100
```

This new value, when propagated through the network, forces C to have a value of 100, and this is registered by the probe on C. Notice that the very same network is being used to compute C given F and to compute F given C. This nondirectionality of computation is the distinguishing feature of constraint-based systems.

Implementing the constraint system

The constraint system is implemented via procedural objects with local state, in a manner very similar to the digital-circuit simulator of section 3.3.4. Although the primitive objects of the constraint system are somewhat more complex, the overall system is simpler, since there is no concern about agendas and logic delays.

The basic operations on connectors are the following:

- `has_value(connector)`: tells whether the connector has a value.
- `get_value(connector)`: returns the connector's current value.
- `set_value(connector, new_value, informant)`: indicates that the informant is requesting the connector to set its value to the new value.
- `forget_value(connector, retractor)`: tells the connector that the retractor is requesting it to forget its value.
- `connect(connector, new_constraint)`: tells the connector to participate in the new constraint.

The connectors communicate with the constraints by means of the functions `inform_about_value`, which tells the given constraint that the connector has a value, and `forget_value`, which tells the constraint that the connector has lost its value.

`Adder` constructs an adder constraint among summand connectors `a1` and `a2` and a `sum` connector. An adder is implemented as a function with local state (the function `me` below):

```
function adder(a1, a2, sum) {
    function process_new_value() {
        if (has_value(a1) && has_value(a2)) {
            set_value(sum, get_value(a1) + get_value(a2), me);
        } else if (has_value(a1) && has_value(sum)) {
            set_value(a2, get_value(sum) - get_value(a1), me);
        } else if (has_value(a2) && has_value(sum)) {
            set_value(a1, get_value(sum) - get_value(a2), me);
        } else {
        }
    }
    function process_forget_value() {
        forget_value(sum, me);
        forget_value(a1, me);
        forget_value(a2, me);
        process_new_value();
    }
    function me(request) {
```

```
        if (request === "I-have-a-value") {
            process_new_value();
        } else if (request === "I-lost-my-value") {
            process_forget_value();
        } else {
            Error("Unknown request in adder", request);
        }
    }
    connect(a1, me);
    connect(a2, me);
    connect(sum, me);
    return me;
}
```

`Adder` connects the new adder to the designated connectors and returns it as its value. The function `me`, which represents the adder, acts as a dispatch to the local functions. The following 'syntax interfaces' (see footnote 25 in section 3.3.4) are used in conjunction with the dispatch:

```
function inform_about_value(constraint) {
    return constraint("I-have-a-value");
}

function inform_about_no_value(constraint) {
    return constraint("I-lost-my-value");
}
```

The adder's local function `process_new_value` is called when the adder is informed that one of its connectors has a value. The adder first checks to see if both `a1` and `a2` have values. If so, it tells `sum` to set its value to the sum of the two addends. The `informant` argument to `set_value` is `me`, which is the adder object itself. If `a1` and `a2` do not both have values, then the adder checks to see if perhaps `a1` and `sum` have values. If so, it sets `a2` to the difference of these two. Finally, if `a2` and `sum` have values, this gives the adder enough information to set `a1`. If the adder is told that one of its connectors has lost a value, it requests that all of its connectors now lose their values. (Only those values that were set by this adder are actually lost.) Then it runs `process_new_value`. The reason for this last step is that one or more connectors may still have a value (that is, a connector may have had a value that was not originally set by the adder), and these values may need to be propagated back through the adder.

A multiplier is very similar to an adder. It will set its `product` to 0 if either of the factors is 0, even if the other factor is not known.

```
function multiplier(m1, m2, product) {
    function process_new_value() {
        if ((has_value(m1) && get_value(m1) === 0)
         || (has_value(m2) && get_value(m2) === 0)) {
            set_value(product, 0, me);
        } else if (has_value(m1) && has_value(m2)) {
```

```
            set_value(product,
                      get_value(m1) * get_value(m2),
                      me);
        } else if (has_value(product) && has_value(m1)) {
            set_value(m2,
                      get_value(product) / get_value(m1),
                      me);
        } else if (has_value(product) && has_value(m2)) {
            set_value(m1,
                      get_value(product) / get_value(m2),
                      me);
        } else {
        }
    }
    function process_forget_value() {
        forget_value(product, me);
        forget_value(m1, me);
        forget_value(m2, me);
        process_new_value();
    }
    function me(request) {
        if (request === "I-have-a-value") {
            process_new_value();
        } else if (request === "I-lost-my-value") {
            process_forget_value();
        } else {
            Error("Unknown request in multiplier", request);
        }
    }
    connect(m1, me);
    connect(m2, me);
    connect(product, me);
    return me;
}
```

A `constant` constructor simply sets the value of the designated connector. Any `"I-have-a-value"` or `"I-lost-my-value"` message sent to the constant box will produce an error.

```
function constant(value, connector) {
    function me(request) {
        Error("Unknown request in constant", request);
    }
    connect(connector, me);
    set_value(connector, value, me);
    return me;
}
```

Finally, a probe prints a message about the setting or unsetting of the designated connector:

```
function probe(name, connector) {
    function print_probe(value) {
        display("Probe: " + name + " = " + value);
    }
    function process_new_value() {
        print_probe(get_value(connector));
    }
    function process_forget_value() {
        print_probe("?");
    }
    function me(request) {
        return request === "I-have-a-value"
               ? process_new_value()
               : request === "I-lost-my-value"
                 ? process_forget_value()
                 : Error("Unknown request in probe",
                         request);
    }
    connect(connector, me);
    return me;
}
```

Representing connectors

A connector is represented as a procedural object with local state variables `value`, the current value of the connector; `informant`, the object that set the connector's value; and `constraints`, a list of the constraints in which the connector participates.

```
function make_connector() {
    let value = false;
    let informant = false;
    let constraints = null;
    function set_my_value(newval, setter) {
        if (!has_value(me)) {
            value = newval;
            informant = setter;
            for_each_except(setter,
                            inform_about_value,
                            constraints);
        } else if (value !== newval) {
            error("Contradiction " +
                  "(" + stringify(value) + ", " +
                      + stringify(newval) + ")");
        } else {
            return "ignored";
        }
    }
```

```
    function forget_my_value(retractor) {
        if (retractor === informant) {
            informant = false;
            for_each_except(retractor,
                            inform_about_no_value,
                            constraints);
        } else {
            return "ignored";
        }
    }
    function connect(new_contraint) {
        if (is_null(member(new_contraint,
                           constraints))) {
            constraints = pair(new_contraint, constraints);
        } else {
        }
        if (has_value(me)) {
            inform_about_value(new_contraint);
        } else {
        }

        return "done";
    }
    function me(request) {
        if (request === "has_value") {
            return informant !== false;
        } else if (request === "value") {
            return value;
        } else if (request === "set_value") {
            return set_my_value;
        } else if (request === "forget") {
            return forget_my_value;
        } else if (request === "connect") {
            return connect;
        } else {
            Error("Unknown operation in connector", request);
        }
    }
    return me;
}
```

The connector's local function `set_my_value` is called when there is a request to set the connector's value. If the connector does not currently have a value, it will set its value and remember as `informant` the constraint that requested the value to be set.[30] Then the connector will notify all of its participating constraints except the constraint that requested the value to be set. This is accomplished using the following iterator, which applies a designated function to

[30]The `setter` might not be a constraint. In our temperature example, we used `user` as the `setter`.

all items in a list except a given one:

```
function for_each_except(exception, fun, list) {
    function loop(items) {
        if (is_null(items)) {
            return "done";
        } else if (head(items) === exception) {
            return loop(tail(items));
        } else {
            fun(head(items));
            return loop(tail(items));
        }
    }
    return loop(list);
}
```

If a connector is asked to forget its value, it runs the local function `forget_my_value`, which first checks to make sure that the request is coming from the same object that set the value originally. If so, the connector informs its associated constraints about the loss of the value.

The local function `connect` adds the designated new constraint to the list of constraints if it is not already in that list. Then, if the connector has a value, it informs the new constraint of this fact.

The connector's function `me` serves as a dispatch to the other internal functions and also represents the connector as an object. The following functions provide a syntax interface for the dispatch:

```
function has_value(connector) {
    return connector("has_value");
}

function get_value(connector) {
    return connector("value");
}

function set_value(connector, new_value, informant) {
    return (connector("set_value"))(new_value, informant);
}

function forget_value(connector, retractor) {
    return (connector("forget"))(retractor);
}

function connect(connector, new_constraint) {
    return (connector("connect"))(new_constraint);
}
```

Exercise 3.33

Using primitive multiplier, adder, and constant constraints, define a function `averager` that takes three connectors `a`, `b`, and `c` as inputs and establishes the constraint that the value of `c` is the average of the values of `a` and `b`.

Exercise 3.34

Louis Reasoner wants to build a squarer, a constraint device with two terminals such that the value of connector `b` on the second terminal will always be the square of the value `a` on the first terminal. He proposes the following simple device made from a multiplier:

```
function squarer(a, b) {
    return multiplier(a, a, b);
}
```

There is a serious flaw in this idea. Explain.

Exercise 3.35

Ben Bitdiddle tells Louis that one way to avoid the trouble in exercise 3.34 is to define a squarer as a new primitive constraint. Fill in the missing portions in Ben's outline for a function to implement such a constraint:

```
function squarer(a, b) {
    function process_new_value() {
        if (has_value(b)) {
            if (get_value(b) < 0) {
                Error("Square less than 0 in squarer",
                      get_value(b));
            } else {
                // alternative1...
            } else {
                // alternative2...
            }
        }
    }
    function process_forget_value() {
        // body1...
    }
    function me(request) {
        // body2...
    }
    // rest of definition
    return me;
}
```

Exercise 3.36

Suppose we evaluate the following sequence of expressions in the program environment:

```
const a = make_connector();
const b = make_connector();
set_value(a, 10, "user");
```

At some time during evaluation of the `set_value`, the following expression from the connector's local function is evaluated:

```
for_each_except(setter, inform_about_value, constraints);
```

Draw an environment diagram showing the environment in which the above expression is evaluated.

Exercise 3.37

The `celsius_fahrenheit_converter` function is cumbersome when compared with a more expression-oriented style of definition, such as

```
function celsius_fahrenheit_converter(x) {
    return cplus(cmul(cdiv(cv(9), cv(5)), x), cv(32));
}
```

Here `cplus` , `cmul` , etc. are the 'constraint' versions of the arithmetic operations. For example, `cplus` takes two connectors as arguments and returns a connector that is related to these by an adder constraint:

```
function cplus(x, y) {
    const z = make_connector();
    adder(x, y, z);
    return z;
}
```

Define analogous functions `cminus` , `cmul` , `cdiv` , and `cv` (constant value) that enable us to define compound constraints as in the converter example above.[31]

[31] The expression-oriented format is convenient because it avoids the need to name the intermediate expressions in a computation. Our original formulation of the constraint language is cumbersome in the same way that many languages are cumbersome when dealing with operations on compound data. For example, if we wanted to compute the product $(a + b) \cdot (c + d)$, where the variables represent vectors, we could work in 'imperative style,' using functions that set the values of designated vector arguments but do not themselves return vectors as values:

```
v_sum('a', 'b', temp1);
v_sum('c', 'd', temp2);
v_prod(temp1, temp2, answer);
```

Alternatively, we could deal with expressions, using functions that return vectors as values, and thus avoid explicitly mentioning `temp1` and `temp2`:

3.4 Concurrency: Time Is of the Essence

We've seen the power of computational objects with local state as tools for modeling. Yet, as Section 3.1.3 warned, this power extracts a price: the loss of referential transparency, giving rise to a thicket of questions about sameness and change, and the need to abandon the substitution model of evaluation in favor of the more intricate environment model.

The central issue lurking beneath the complexity of state, sameness, and change is that by introducing assignment we are forced to admit *time* into our computational models. Before we introduced assignment, all our programs were timeless, in the sense that any expression that has a value always has the same value. In contrast, recall the example of modeling withdrawals from a bank account and returning the resulting balance, introduced at the beginning of Section 3.1.1:

```
withdraw(25); // output: 75
```

```
withdraw(25); // output: 50
```

Here successive evaluations of the same expression yield different values. This behavior arises from the fact that the execution of assignment statements (in this case, assignments to the variable `balance`) delineates *moments in time* when values change. The result of evaluating an expression depends not only on the expression itself, but also on whether the evaluation occurs before or after these moments. Building models in terms of computational objects with local state forces us to confront time as an essential concept in programming.

We can go further in structuring computational models to match our perception of the physical world. Objects in the world do not change one at a time in sequence. Rather we perceive them as acting *concurrently*—all at once. So it is often natural to model systems as collections of computational processes that execute concurrently. Just as we can make our programs modular by organizing models in terms of objects with separate local state, it is often appropriate to divide computational models into parts that evolve separately and concurrently. Even if the programs are to be executed on a sequential computer, the practice of writing programs as if they were to be executed concurrently forces the programmer to avoid inessential timing

---

```
const answer = v_prod(v_sum('a', 'b'), v_sum('c', 'd'));
```

Since JavaScript allows us to return compound objects as values of functions, we can transform our imperative-style constraint language into an expression-oriented style as shown in this exercise. In languages that are impoverished in handling compound objects, such as Algol, Basic, and Pascal (unless one explicitly uses Pascal pointer variables), one is usually stuck with the imperative style when manipulating compound objects. Given the advantage of the expression-oriented format, one might ask if there is any reason to have implemented the system in imperative style, as we did in this section. One reason is that the non-expression-oriented constraint language provides a handle on constraint objects (e.g., the value of the `adder` function) as well as on connector objects. This is useful if we wish to extend the system with new operations that communicate with constraints directly rather than only indirectly via operations on connectors. Although it is easy to implement the expression-oriented style in terms of the imperative implementation, it is very difficult to do the converse.

constraints and thus makes programs more modular.

In addition to making programs more modular, concurrent computation can provide a speed advantage over sequential computation. Sequential computers execute only one operation at a time, so the amount of time it takes to perform a task is proportional to the total number of operations performed.[32]

However, if it is possible to decompose a problem into pieces that are relatively independent and need to communicate only rarely, it may be possible to allocate pieces to separate computing processors, producing a speed advantage proportional to the number of processors available.

Unfortunately, the complexities introduced by assignment become even more problematic in the presence of concurrency. The fact of concurrent execution, either because the world operates in parallel or because our computers do, entails additional complexity in our understanding of time.

3.4.1 The Nature of Time in Concurrent Systems

On the surface, time seems straightforward. It is an ordering imposed on events.[33] For any events A and B, either A occurs before B, A and B are simultaneous, or A occurs after B. For instance, returning to the bank account example, suppose that Peter withdraws \$10 and Paul withdraws \$25 from a joint account that initially contains \$100, leaving \$65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either \$100 → \$90 → \$65\$100 → \$75 → \$65. In a computer implementation of the banking system, this changing sequence of balances could be modeled by successive assignments to a variable `balance`.

In complex situations, however, such a view can be problematic. Suppose that Peter and Paul, and other people besides, are accessing the same bank account through a network of banking machines distributed all over the world. The actual sequence of balances in the account will depend critically on the detailed timing of the accesses and the details of the communication among the machines.

This indeterminacy in the order of events can pose serious problems in the design of concurrent systems. For instance, suppose that the withdrawals made by Peter and Paul are implemented as two separate processes sharing a common variable `balance`, each process specified by the function given in Section 3.1.1:

[32] Most real processors actually execute a few operations at a time, following a strategy called *pipelining*. Although this technique greatly improves the effective utilization of the hardware, it is used only to speed up the execution of a sequential instruction stream, while retaining the behavior of the sequential program.

[33] To quote some graffiti seen on a Cambridge building wall: 'Time is a device that was invented to keep everything from happening at once.'

```
function withdraw(amount) {
    if (balance >= amount) {
        balance = balance - amount;
        return balance;
    } else {
        return "Insufficient funds";
    }
}
```

If the two processes operate independently, then Peter might test the balance and attempt to withdraw a legitimate amount. However, Paul might withdraw some funds in between the time that Peter checks the balance and the time Peter completes the withdrawal, thus invalidating Peter's test.

Things can be worse still. Consider the expression

```
balance = balance - amount;
```

executed as part of each withdrawal process. This consists of three steps: (1) accessing the value of the `balance` variable; (2) computing the new balance; (3) setting `balance` to this new value. If Peter and Paul's withdrawals execute this statement concurrently, then the two withdrawals might interleave the order in which they access `balance` and set it to the new value.

The timing diagram in Figure 3.29 depicts an order of events where `balance` starts at 100, Peter withdraws 10, Paul withdraws 25, and yet the final value of `balance` is 75. As shown in the diagram, the reason for this anomaly is that Paul's assignment of 75 to `balance` is made under the assumption that the value of `balance` to be decremented is 100. That assumption, however, became invalid when Peter changed `balance` to 90. This is a catastrophic failure for the banking system, because the total amount of money in the system is not conserved. Before the transactions, the total amount of money was $100. Afterwards, Peter has $10, Paul has $25, and the bank has $75.[34]

The general phenomenon illustrated here is that several processes may share a common state variable. What makes this complicated is that more than one process may be trying to manipulate the shared state at the same time. For the bank account example, during each transaction, each customer should be able to act as if the other customers did not exist. When a customer changes the balance in a way that depends on the balance, he must be able to assume that, just before the moment of change, the balance is still what he thought it was.

[34] An even worse failure for this system could occur if the two assignment statements attempt to change the balance simultaneously, in which case the actual data appearing in memory might end up being a random combination of the information being written by the two processes. Most computers have interlocks on the primitive memory-write operations, which protect against such simultaneous access. Even this seemingly simple kind of protection, however, raises implementation challenges in the design of multiprocessing computers, where elaborate *cache-coherence* protocols are required to ensure that the various processors will maintain a consistent view of memory contents, despite the fact that data may be replicated ('cached') among the different processors to increase the speed of memory access.

JavaScript and concurrency

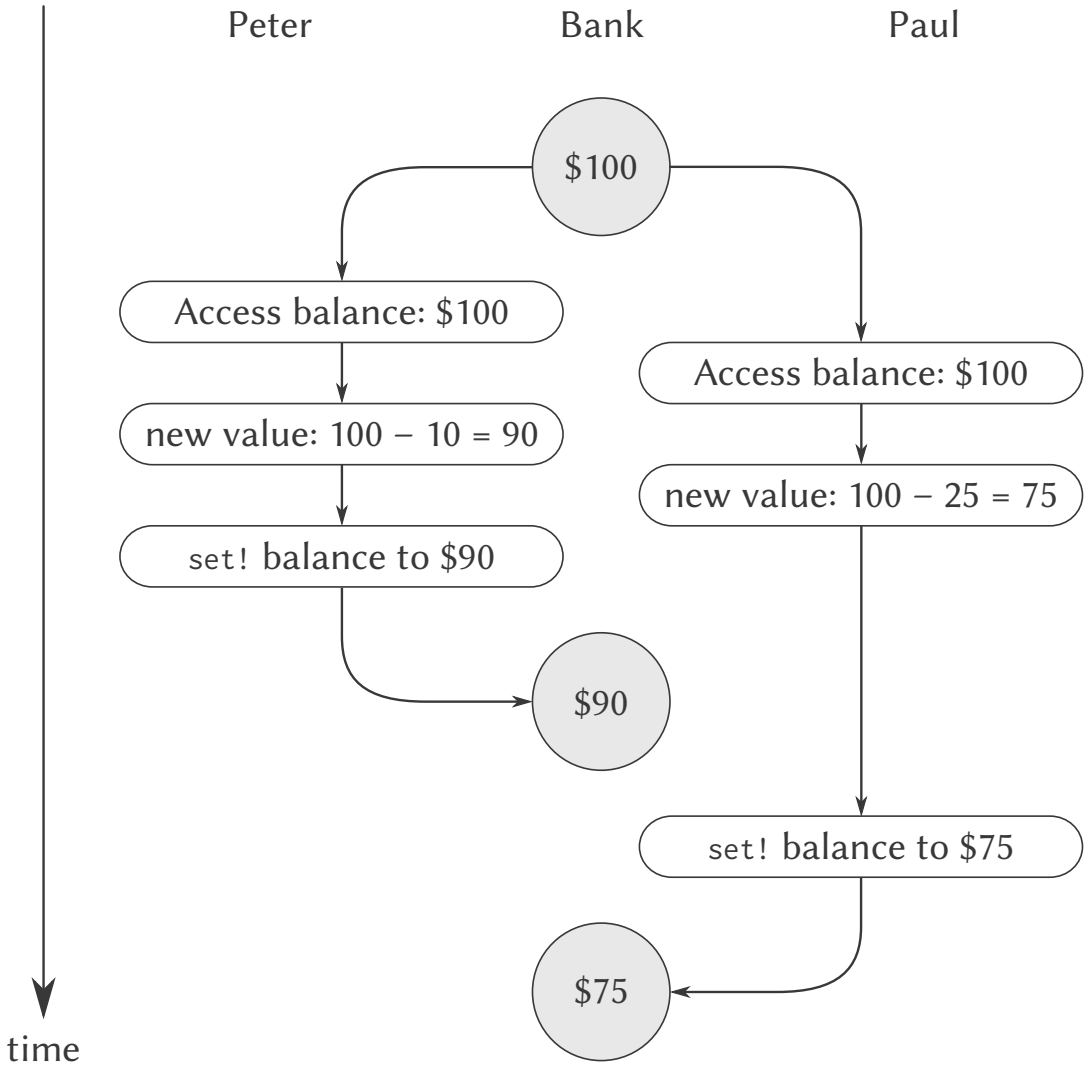
In its initial design, JavaScript did not allow for two processes to apply functions such as `withdraw` concurrently. In fact, the concurrency model of the language enforced strict sequential execution of activities resulting from *events*, with the use of an *event queue*. In the early 2000s, multicore computers became common and around 2010, the JavaScript designers responded with the introduction of concurrent processes via the concept of *web workers*. As of 2019, most internet browsers support this feature. As originally conceived, web workers were not able to share data such as the variable `balance` above. However, a shared data structure called `SharedArrayBuffer` is included in the latest ECMAScript specification. Using `SharedArrayBuffer`, it is possible to program a `withdraw` function as described above.[35]

Correct behavior of concurrent programs

The above example typifies the subtle bugs that can creep into concurrent programs. The root of this complexity lies in the assignments to variables that are shared among the different processes. We already know that we must be careful in writing programs that use assignment, because the results of a computation depend on the order in which the assignments occur.[36]

With concurrent processes we must be especially careful about assignments, because we may not be able to control the order of the assignments made by the different processes. If several such changes might be made concurrently (as with two depositors accessing a joint account) we need some way to ensure that our system behaves correctly. For example, in the case of withdrawals from a joint bank account, we must ensure that money is conserved. To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.

[35] As of 2019, browsers differ in their support for `SharedArrayBuffer` objects.
[36] The factorial program in section 3.1.3 illustrates this for a single sequential process.



(a)

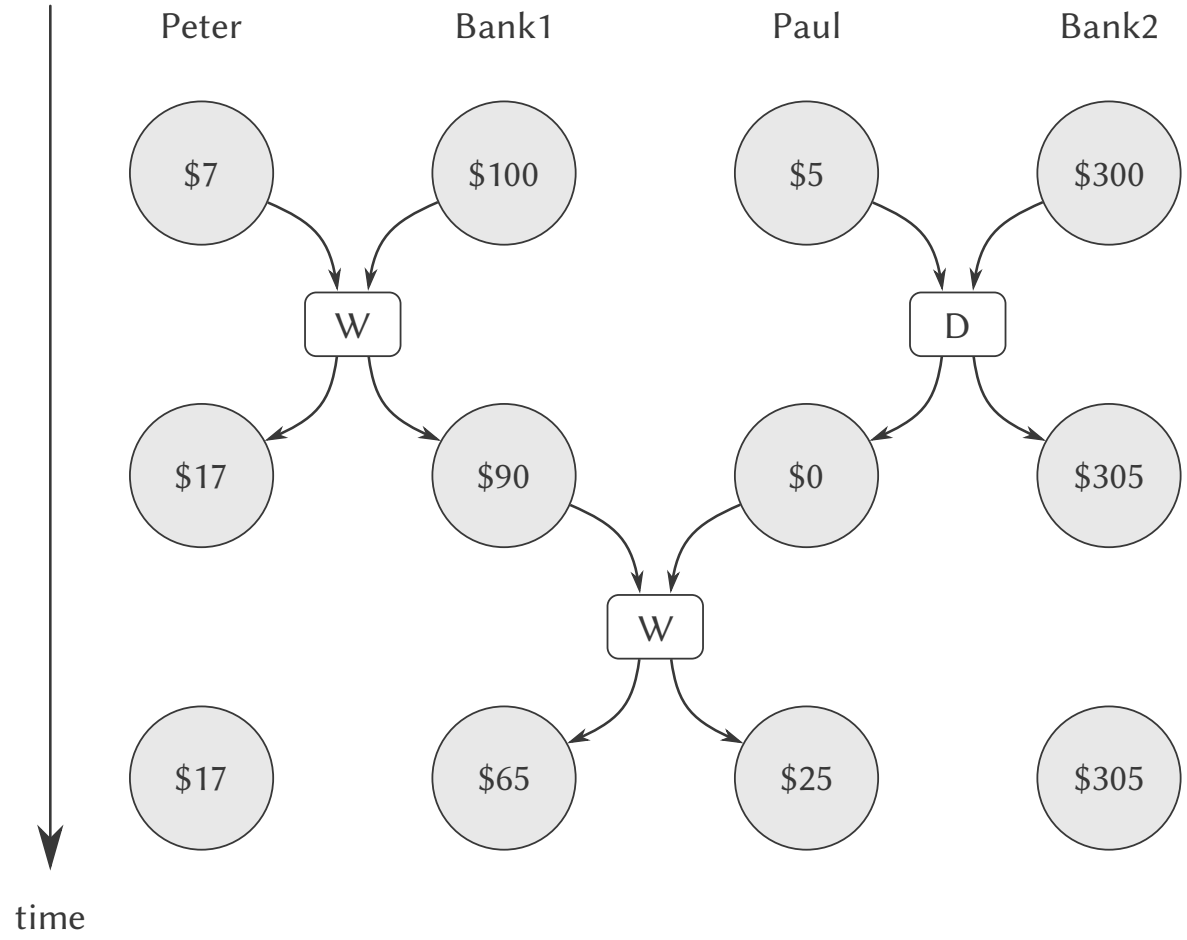
Figure 3.29: Timing diagram showing how interleaving the order of events in two banking withdrawals can lead to an incorrect final balance.

One possible restriction on concurrency would stipulate that no two operations that change any shared state variables can occur at the same time. This is an extremely stringent requirement. For distributed banking, it would require the system designer to ensure that only one transaction could proceed at a time. This would be both inefficient and overly conservative. Figure 3.30 shows Peter and Paul sharing a bank account, where Paul has a private account as well. The diagram illustrates two withdrawals from the shared account (one by Peter and one by Paul) and a deposit to Paul's private account.[37]

The two withdrawals from the shared account must not be concurrent (since both access and update the same account), and Paul's deposit and withdrawal must not be concurrent (since

[37]The columns show the contents of Peter's wallet, the joint account (in Bank1), Paul's wallet, and Paul's private account (in Bank2), before and after each withdrawal (W) and deposit (D). Peter withdraws $10 from Bank1; Paul deposits $5 in Bank2, then withdraws $25 from Bank1.

both access and update the amount in Paul's wallet). But there should be no problem permitting Paul's deposit to his private account to proceed concurrently with Peter's withdrawal from the shared account.



(a)

Figure 3.30: Concurrent deposits and withdrawals from a joint account in Bank1 and a private account in Bank2.

A less stringent restriction on concurrency would ensure that a concurrent system produces the same result as if the processes had run sequentially in some order. There are two important aspects to this requirement. First, it does not require the processes to actually run sequentially, but only to produce results that are the same *as if* they had run sequentially. For the example in Figure 3.30, the designer of the bank account system can safely allow Paul's deposit and Peter's withdrawal to happen concurrently, because the net result will be the same as if the two operations had happened sequentially. Second, there may be more than one possible 'correct' result produced by a concurrent program, because we require only that the result be the same as for *some* sequential order. For example, suppose that Peter and Paul's joint account starts out with $100, and Peter deposits $40 while Paul concurrently withdraws half the money in the account. Then sequential execution could result in the account balance being either $70

or $90 (see exercise 3.38).[38]

There are still weaker requirements for correct execution of concurrent programs. A program for simulating diffusion (say, the flow of heat in an object) might consist of a large number of processes, each one representing a small volume of space, that update their values concurrently. Each process repeatedly changes its value to the average of its own value and its neighbors' values. This algorithm converges to the right answer independent of the order in which the operations are done; there is no need for any restrictions on concurrent use of the shared values.

Exercise 3.38

Suppose that Peter, Paul, and Mary share a joint bank account that initially contains $100. Concurrently, Peter deposits $10, Paul withdraws $20, and Mary withdraws half the money in the account, by executing the following commands:

a. List all the different possible values for `balance` after these three transactions have been completed, assuming that the banking system forces the three processes to run sequentially in some order.

b. What are some other values that could be produced if the system allows the processes to be interleaved? Draw timing diagrams like the one in figure 3.29 to explain how these values can occur.

3.4.2 Mechanisms for Controlling Concurrency

We've seen that the difficulty in dealing with concurrent processes is rooted in the need to consider the interleaving of the order of events in the different processes. For example, suppose we have two processes, one with three ordered events (a, b, c) and one with three ordered events (x, y, z). If the two processes run concurrently, with no constraints on how their execution is interleaved, then there are 20 different possible orderings for the events that are consistent with the individual orderings for the two processes:

$$
\begin{array}{cccc}
(a,b,c,x,y,z) & (a,x,b,y,c,z) & (x,a,b,c,y,z) & (x,a,y,z,b,c) \\
(a,b,x,c,y,z) & (a,x,b,y,z,c) & (x,a,b,y,c,z) & (x,y,a,b,c,z) \\
(a,b,x,y,c,z) & (a,x,y,b,c,z) & (x,a,b,y,z,c) & (x,y,a,b,z,c) \\
(a,b,x,y,z,c) & (a,x,y,b,z,c) & (x,a,y,b,c,z) & (x,y,a,z,b,c) \\
(a,x,b,c,y,z) & (a,x,y,z,b,c) & (x,a,y,b,z,c) & (x,y,z,a,b,c)
\end{array}
$$

[38] A more formal way to express this idea is to say that concurrent programs are inherently *nondeterministic*. That is, they are described not by single-valued functions, but by functions whose results are sets of possible values.

As programmers designing this system, we would have to consider the effects of each of these 20 orderings and check that each behavior is acceptable. Such an approach rapidly becomes unwieldy as the numbers of processes and events increase.

A more practical approach to the design of concurrent systems is to devise general mechanisms that allow us to constrain the interleaving of concurrent processes so that we can be sure that the program behavior is correct. Many mechanisms have been developed for this purpose. In this section, we describe one of them, the *serializer*.

Serializing access to shared state

Serialization implements the following idea: Processes will execute concurrently, but there will be certain collections of functions that cannot be executed concurrently. More precisely, serialization creates distinguished sets of functions such that only one execution of a function in each serialized set is permitted to happen at a time. If some function in the set is being executed, then a process that attempts to execute any function in the set will be forced to wait until the first execution has finished.

We can use serialization to control access to shared variables. For example, if we want to update a shared variable based on the previous value of that variable, we put the access to the previous value of the variable and the assignment of the new value to the variable in the same function. We then ensure that no other function that assigns to the variable can run concurrently with this function by serializing all of these functions with the same serializer. This guarantees that the value of the variable cannot be changed between an access and the corresponding assignment.

Serializers in JavaScript

To make the above mechanism more concrete, suppose that we have extended JavaScript to include a function called `parallel_execute`:

`parallel_execute( f1, f2, ..., fk )`

Each f must be a function of no arguments. The function `parallel_execute` creates a separate process for each f, which applies f (to no arguments). These processes all run concurrently.[39]

As an example of how this is used, consider

```
let x = 10;

parallel_execute( () => { x = x * x; },
                  () => { x = x + 1; } );
```

[39]The function `parallel_execute` is not part of the JavaScript standard, but it can be implemented using the `SharedArrayBuffer` feature mentioned in section 3.4.1.

This creates two concurrent processes—P_1, which sets x to x times x, and P_2, which increments x. After execution is complete, x will be left with one of five possible values, depending on the interleaving of the events of P_1 and P_2:

- 101: P_1 sets x to 100 and then P_2 increments x to 101.
- 121: P_2 increments x to 11 and then P_1 sets x to x times x.
- 110: P_2 changes x from 10 to 11 between the two times that P_1 accesses the value of x during the evaluation of `x * x`.
- 11: P_2 accesses x, then P_1 sets x to 100, then P_2 sets x.
- 100: P_1 accesses x (twice), then P_2 sets x to 11, then P_1 sets x.

We can constrain the concurrency by using serialized functions, which are created by *serializers*. Serializers are constructed by `make_serializer`, whose implementation is given below. A serializer takes a function as argument and returns a serialized function that behaves like the original function. All calls to a given serializer return serialized functions in the same set.

Thus, in contrast to the example above, executing

```
let x = 10;

const s = make_serializer();

parallel_execute(s( () => { x = x * x; }),
                 s( () => { x = x + 1; } ));
```

can produce only two possible values for x, 101 or 121. The other possibilities are eliminated, because the execution of P_1 and P_2 cannot be interleaved.

Here is a version of the `make_account` function from section 3.1.1, where the deposits and withdrawals have been serialized:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
```

```
    const protector = make_serializer();
    function dispatch(m) {
        return m === "withdraw"
               ? protector(withdraw)
               : m === "deposit"
                 ? protector(deposit)
                 : m === "balance"
                   ? balance
                   : Error("Unknown request in make_account",
                           m);
    }
    return dispatch;
}
```

With this implementation, two processes cannot be withdrawing from or depositing into a single account concurrently. This eliminates the source of the error illustrated in figure 3.29, where Peter changes the account balance between the times when Paul accesses the balance to compute the new value and when Paul actually performs the assignment. On the other hand, each account has its own serializer, so that deposits and withdrawals for different accounts can proceed concurrently.

Exercise 3.39

Which of the five possibilities in the parallel execution shown above remain if we instead serialize execution as follows:

```
let x = 10;

const s = make_serializer();

parallel_execute( () => { x = s( () => x * x ); },
                  s( () => { x = x + 1; } ) );
```

Exercise 3.40

Give all possible values of x that can result from executing

```
let x = 10;

parallel_execute( () => { x = x * x; },
                  () => { x = x * x * x; } );
```

Which of these possibilities remain if we instead use serialized functions:

```
let x = 10;
```

```
const s = make_serializer();

parallel_execute( s( () => x = x * x ) ,
                  s( () => x = x * x * x ) )
```

Exercise 3.41

Ben Bitdiddle worries that it would be better to implement the bank account as follows (where the commented line has been changed):

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    const protected_ = make_serializer();
    function dispatch(m) {
        return m === "withdraw"
               ? protected_(withdraw)
                 : m === "deposit"
                   ? protected_(deposit)
                   : m === "balance"
                     ? protected_( () => balance )() // serialized
                     : error("Unknown request in make_account",
                             m);
                    }
                }
                return dispatch;
            }
```

because allowing unserialized access to the bank balance can result in anomalous behavior. Do you agree? Is there any scenario that demonstrates Ben's concern?

Exercise 3.42

Ben Bitdiddle suggests that it's a waste of time to create a new serialized function in response to every `withdraw` and `deposit` message. He says that `make_account` could be changed so that the calls to `protected_` are done outside the `dispatch` function. That is, an account would

return the same serialized function (which was created at the same time as the account) each time it is asked for a withdrawal function.

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    const protected_ = make_serializer();
    const protected_withdraw = protected_(withdraw);
    const protected_deposit = protected_(deposit);
    function dispatch(m) {
        return m === "withdraw"
               ? protected_withdraw
               : m === "deposit"
                 ? protected_deposit
                 : m === "balance"
                   ? balance
                   : error("Unknown request in make_account",
                           m);
    }
    return dispatch;
}
```

Is this a safe change to make? In particular, is there any difference in what concurrency is allowed by these two versions of `make_account` ?

Complexity of using multiple shared resources

Serializers provide a powerful abstraction that helps isolate the complexities of concurrent programs so that they can be dealt with carefully and (hopefully) correctly. However, while using serializers is relatively straightforward when there is only a single shared resource (such as a single bank account), concurrent programming can be treacherously difficult when there are multiple shared resources.

To illustrate one of the difficulties that can arise, suppose we wish to swap the balances in two bank accounts. We access each account to find the balance, compute the difference between the balances, withdraw this difference from one account, and deposit it in the other account.

We could implement this as follows:[40]

```
function exchange(account1, account2) {
    const difference = account1("balance") - account2("balance");
    account1("withdraw")(difference);
    account2("deposit")(difference);
}
```

This function works well when only a single process is trying to do the exchange. Suppose, however, that Peter and Paul both have access to accounts a_1, a_2, and a_3, and that Peter exchanges a_1 and a_2 while Paul concurrently exchanges a_1 and a_3. Even with account deposits and withdrawals serialized for individual accounts (as in the `make_account` function shown above in this section), `exchange` can still produce incorrect results. For example, Peter might compute the difference in the balances for a_1 and a_2, but then Paul might change the balance in a_1 before Peter is able to complete the exchange.[41] For correct behavior, we must arrange for the `exchange` function to lock out any other concurrent accesses to the accounts during the entire time of the exchange.

One way we can accomplish this is by using both accounts' serializers to serialize the entire `exchange` function. To do this, we will arrange for access to an account's serializer. Note that we are deliberately breaking the modularity of the bank-account object by exposing the serializer. The following version of `make_account` is identical to the original version given in Section 3.1.1, except that a serializer is provided to protect the balance variable, and the serializer is exported via message passing:

```
function make_account_and_serializer(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
        } else {
            "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    const balance_serializer = make_serializer();
    function dispatch(m) {
        if (m === "withdraw") {
            return withdraw;
        } else if (m === "deposit") {
```

[40] We have simplified `exchange` by exploiting the fact that our `deposit` message accepts negative amounts. (This is a serious bug in our banking system!)

[41] If the account balances start out as \$10, \$20, and \$30, then after any number of concurrent exchanges, the balances should still be \$10, \$20, and \$30 in some order. Serializing the deposits to individual accounts is not sufficient to guarantee this. See exercise 3.43.

```
            return deposit;
        } else if (m === "balance") {
            return balance;
        } else if (m === "serializer") {
            return balance_serializer;
        } else {
            return "Unknown request - - MAKE-ACCOUNT";
        }
    }
    return dispatch;
}
```

We can use this to do serialized deposits and withdrawals. However, unlike our earlier serialized account, it is now the responsibility of each user of bank-account objects to explicitly manage the serialization, for example as follows:[42]

```
function deposit(account, amount) {
    const s = account("serializer");
    const d = account("deposit");
    s(d(amount));
}
```

Exporting the serializer in this way gives us enough flexibility to implement a serialized exchange program. We simply serialize the original exchange function with the serializers for both accounts:

```
function serialized_exchange(account1, account2) {
    const serializer1 = account1("serializer");
    const serializer2 = account2("serializer");
    serializer1(serializer2(exchange))(account1, account2);
}
```

Exercise 3.43

Suppose that the balances in three accounts start out as $10, $20, and $30, and that multiple processes run, exchanging the balances in the accounts. Argue that if the processes are run sequentially, after any number of concurrent exchanges, the account balances should be $10, $20, and $30 in some order. Draw a timing diagram like the one in Figure 3.29 to show how this condition can be violated if the exchanges are implemented using the first version of the account-exchange program in this section. On the other hand, argue that even with this exchange program, the sum of the balances in the accounts will be preserved. Draw a timing diagram to show how even this condition would be violated if we did not serialize the transactions on individual accounts.

[42] Exercise 3.45 investigates why deposits and withdrawals are no longer automatically serialized by the account.

Exercise 3.44

Consider the problem of transferring an amount from one account to another. Ben Bitdiddle claims that this can be accomplished with the following function, even if there are multiple people concurrently transferring money among multiple accounts, using any account mechanism that serializes deposit and withdrawal transactions, for example, the version of `make_account` in the text above.

```
function transfer(from_account, to_account, amount) {
    from_account("withdraw")(amount);
    to_account("deposit")(amount);
}
```

Louis Reasoner claims that there is a problem here, and that we need to use a more sophisticated method, such as the one required for dealing with the exchange problem. Is Louis right? If not, what is the essential difference between the transfer problem and the exchange problem? (You should assume that the balance in `from_account` is at least `amount`.)

Exercise 3.45

Louis Reasoner thinks our bank-account system is unnecessarily complex and error-prone now that deposits and withdrawals aren't automatically serialized. He suggests that `make_account_and_serializer` should have exported the serializer (for use by such functions as `serialized_exchange`) in addition to (rather than instead of) using it to serialize accounts and deposits as `make_account` did. He proposes to redefine accounts as follows:

```
function make_account_and_serializer(balance) {
    function withdraw(amount) {
        if (balance > amount) {
            balance = balance - amount;
        } else {
            "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }
    const balance_serializer = make_serializer();
    function dispatch(m) {
        return m === "withdraw"
               ? balance_serializer(withdraw)
               : m === "deposit"
                 ? balance_serializer(deposit)
                 : m === "balance"
                   ? balance
```

```
                    : m === "serializer"
                      ? balance_serializer
                      : error("Unknown request in make_account",
                              m);
    }
    return dispatch;
}
```

Then deposits are handled as with the original `make_account`:

```
function deposit(account, amount) {
    account("deposit")(amount);
}
```

Explain what is wrong with Louis's reasoning. In particular, consider what happens when `serialized_exchange` is called.

Implementing serializers

We implement serializers in terms of a more primitive synchronization mechanism called a *mutex*. A mutex is an object that supports two operations—the mutex can be *acquired*, and the mutex can be *released*. Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released.[43] In our implementation, each serializer has an associated mutex. Given a function `p`, the serializer returns a function that acquires the mutex, runs `p`, and then releases the mutex. This ensures that only one of the functions produced by the serializer can be running at once, which is precisely the serialization property that we need to guarantee.

```
function make_serializer() {
    const mutex = make_mutex();
    return p => {
        function serialized_p(args) {
            mutex("acquire");
            const val = p(args);
            mutex("release");
            return val;
        }
        return serialized_p;
    };
}
```

[43] The term 'mutex' is an abbreviation for *mutual exclusion*. The general problem of arranging a mechanism that permits concurrent processes to safely share resources is called the mutual exclusion problem. Our mutex is a simple variant of the *semaphore* mechanism (see exercise 3.47), which was introduced in the 'THE' Multiprogramming System developed at the Technological University of Eindhoven and named for the university's initials in Dutch (Dijkstra 1968a). The acquire and release operations were originally called P and V, from the Dutch words *passeren* (to pass) and *vrijgeven* (to release), in reference to the semaphores used on railroad systems. Dijkstra's classic exposition (1968b) was one of the first to clearly present the issues of concurrency control, and showed how to use semaphores to handle a variety of concurrency problems.

The mutex is a mutable object (here we'll use a one-element list, which we'll refer to as a *cell*) that can hold the value true or false. When the value is false, the mutex is available to be acquired. When the value is true, the mutex is unavailable, and any process that attempts to acquire the mutex must wait.

Our mutex constructor `make_mutex` begins by initializing the cell contents to false. To acquire the mutex, we test the cell. If the mutex is available, we set the cell contents to true and proceed. Otherwise, we wait in a loop, attempting to acquire over and over again, until we find that the mutex is available.[44] To release the mutex, we set the cell contents to false.

```
function make_mutex() {
    const cell = list(false);
    function the_mutex(m) {
        return m === "aquire"
               ? ( test_and_set(cell)
                   ? the_mutex("aquire") // retry
                   : true )
               : m === "release"
                 ? clear(cell)
                 : error("Unknown request in mutex",
                         m);
    }
    return the_mutex;
}
function clear(cell) {
    set_head(cell, false);
}
```

The function `test_and_set` tests the cell and returns the result of the test. In addition, if the test was false, `test_and_set` sets the cell contents to true before returning false. We can express this behavior as the following function:

```
function test_and_set(cell) {
    if (head(cell)) {
        return true;
    } else {
        set_head(cell, true);
        return false;
    }
}
```

However, this implementation of `test_and_set` does not suffice as it stands. There is a crucial subtlety here, which is the essential place where concurrency control enters the system: The `test_and_set` operation must be performed *atomically*. That is, we must guarantee that, once

[44] In most time-shared operating systems, processes that are blocked by a mutex do not waste time 'busy-waiting' as above. Instead, the system schedules another process to run while the first is waiting, and the blocked process is awakened when the mutex becomes available.

a process has tested the cell and found it to be false, the cell contents will actually be set to true before any other process can test the cell. If we do not make this guarantee, then the mutex can fail in a way similar to the bank-account failure in Figure 3.29. (See exercise 3.46.)

The actual implementation of `test_and_set` depends on the details of how our system runs concurrent processes. For example, we might be executing concurrent processes on a sequential processor using a time-slicing mechanism that cycles through the processes, permitting each process to run for a short time before interrupting it and moving on to the next process. In that case, `test_and_set` can work by disabling time slicing during the testing and setting. Alternatively, multiprocessing computers provide instructions that support atomic operations directly in hardware.[45]

Exercise 3.46

Suppose that we implement `test_and_set` using an ordinary function as shown in the text, without attempting to make the operation atomic. Draw a timing diagram like the one in figure 3.29 to demonstrate how the mutex implementation can fail by allowing two processes to acquire the mutex at the same time.

Exercise 3.47

A semaphore (of size n) is a generalization of a mutex. Like a mutex, a semaphore supports acquire and release operations, but it is more general in that up to n processes can acquire it concurrently. Additional processes that attempt to acquire the semaphore must wait for release operations. Give implementations of semaphores

a. in terms of mutexes

b. in terms of atomic `test_and_set` operations.

[45]There are many variants of such instructions—including test-and-set, test-and-clear, swap, compare-and-exchange, load-reserve, and store-conditional—whose design must be carefully matched to the machine's processor–memory interface. One issue that arises here is to determine what happens if two processes attempt to acquire the same resource at exactly the same time by using such an instruction. This requires some mechanism for making a decision about which process gets control. Such a mechanism is called an *arbiter*. Arbiters usually boil down to some sort of hardware device. Unfortunately, it is possible to prove that one cannot physically construct a fair arbiter that works 100% of the time unless one allows the arbiter an arbitrarily long time to make its decision. The fundamental phenomenon here was originally observed by the fourteenth-century French philosopher Jean Buridan in his commentary on Aristotle's *De caelo*. Buridan argued that a perfectly rational dog placed between two equally attractive sources of food will starve to death, because it is incapable of deciding which to go to first.

Deadlock

Now that we have seen how to implement serializers, we can see that account exchanging still has a problem, even with the `serialized_exchange` function above. Imagine that Peter attempts to exchange a_1 with a_2 while Paul concurrently attempts to exchange a_2 with a_1. Suppose that Peter's process reaches the point where it has entered a serialized function protecting a_1 and, just after that, Paul's process enters a serialized function protecting a_2. Now Peter cannot proceed (to enter a serialized function protecting a_2) until Paul exits the serialized function protecting a_2. Similarly, Paul cannot proceed until Peter exits the serialized function protecting a_1. Each process is stalled forever, waiting for the other. This situation is called a *deadlock*. Deadlock is always a danger in systems that provide concurrent access to multiple shared resources.

One way to avoid the deadlock in this situation is to give each account a unique identification number and rewrite `serialized_exchange` so that a process will always attempt to enter a function protecting the lowest-numbered account first. Although this method works well for the exchange problem, there are other situations that require more sophisticated deadlock-avoidance techniques, or where deadlock cannot be avoided at all. (See exercises 3.48 and 3.49.)[46]

Exercise 3.48

Explain in detail why the deadlock-avoidance method described above, (i.e., the accounts are numbered, and each process attempts to acquire the smaller-numbered account first) avoids deadlock in the exchange problem. Rewrite `serialized_exchange` to incorporate this idea. (You will also need to modify `make_account` so that each account is created with a number, which can be accessed by sending an appropriate message.)

Exercise 3.49

Give a scenario where the deadlock-avoidance mechanism described above does not work. (Hint: In the exchange problem, each process knows in advance which accounts it will need to get access to. Consider a situation where a process must get access to some shared resources before it can know which additional shared resources it will require.)

[46] The general technique for avoiding deadlock by numbering the shared resources and acquiring them in order is due to Havender (1968). Situations where deadlock cannot be avoided require *deadlock-recovery* methods, which entail having processes 'back out' of the deadlocked state and try again. Deadlock-recovery mechanisms are widely used in database management systems, a topic that is treated in detail in Gray and Reuter 1993.

Concurrency, time, and communication

We've seen how programming concurrent systems requires controlling the ordering of events when different processes access shared state, and we've seen how to achieve this control through judicious use of serializers. But the problems of concurrency lie deeper than this, because, from a fundamental point of view, it's not always clear what is meant by 'shared state.'

Mechanisms such as `test_and_set` require processes to examine a global shared flag at arbitrary times. This is problematic and inefficient to implement in modern high-speed processors, where due to optimization techniques such as pipelining and cached memory, the contents of memory may not be in a consistent state at every instant. In contemporary multiprocessing systems, therefore, the serializer paradigm is being supplanted by new approaches to concurrency control.[47]

The problematic aspects of shared state also arise in large, distributed systems. For instance, imagine a distributed banking system where individual branch banks maintain local values for bank balances and periodically compare these with values maintained by other branches. In such a system the value of 'the account balance' would be undetermined, except right after synchronization. If Peter deposits money in an account he holds jointly with Paul, when should we say that the account balance has changed—when the balance in the local branch changes, or not until after the synchronization? And if Paul accesses the account from a different branch, what are the reasonable constraints to place on the banking system such that the behavior is 'correct'? The only thing that might matter for correctness is the behavior observed by Peter and Paul individually and the 'state' of the account immediately after synchronization. Questions about the 'real' account balance or the order of events between synchronizations may be irrelevant or meaningless.[48]

The basic phenomenon here is that synchronizing different processes, establishing shared state, or imposing an order on events requires communication among the processes. In essence, any notion of time in concurrency control must be intimately tied to communication.[49] It is intriguing that a similar connection between time and communication also arises in the Theory of Relativity, where the speed of light (the fastest signal that can be used to synchronize

[47] One such alternative to serialization is called *barrier synchronization*. The programmer permits concurrent processes to execute as they please, but establishes certain synchronization points ('barriers') through which no process can proceed until all the processes have reached the barrier. Modern processors provide machine instructions that permit programmers to establish synchronization points at places where consistency is required. The PowerPC™, for example, includes for this purpose two instructions called SYNC and EIEIO (Enforced In-order Execution of Input/Output).

[48] This may seem like a strange point of view, but there are systems that work this way. International charges to credit-card accounts, for example, are normally cleared on a per-country basis, and the charges made in different countries are periodically reconciled. Thus the account balance may be different in different countries.

[49] For distributed systems, this perspective was pursued by Lamport (1978), who showed how to use communication to establish 'global clocks' that can be used to establish orderings on events in distributed systems.

events) is a fundamental constant relating time and space. The complexities we encounter in dealing with time and state in our computational models may in fact mirror a fundamental complexity of the physical universe.

3.5 Streams

We've gained a good understanding of assignment as a tool in modeling, as well as an appreciation of the complex problems that assignment raises. It is time to ask whether we could have gone about things in a different way, so as to avoid some of these problems. In this section, we explore an alternative approach to modeling state, based on data structures called *streams*. As we shall see, streams can mitigate some of the complexity of modeling state.

Let's step back and review where this complexity comes from. In an attempt to model real-world phenomena, we made some apparently reasonable decisions: We modeled real-world objects with local state by computational objects with local variables. We identified time variation in the real world with time variation in the computer. We implemented the time variation of the states of the model objects in the computer with assignments to the local variables of the model objects.

Is there another approach? Can we avoid identifying time in the computer with time in the modeled world? Must we make the model change with time in order to model phenomena in a changing world? Think about the issue in terms of mathematical functions. We can describe the time-varying behavior of a quantity x as a function of time $x(t)$. If we concentrate on x instant by instant, we think of it as a changing quantity. Yet if we concentrate on the entire time history of values, we do not emphasize change—the function itself does not change.[50]

If time is measured in discrete steps, then we can model a time function as a (possibly infinite) sequence. In this section, we will see how to model change in terms of sequences that represent the time histories of the systems being modeled. To accomplish this, we introduce new data structures called *streams*. From an abstract point of view, a stream is simply a sequence. However, we will find that the straightforward implementation of streams as lists (as in section 2.1.1) doesn't fully reveal the power of stream processing. As an alternative, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) sequences as streams.

Stream processing lets us model systems that have state without ever using assignment or mutable data. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment. On the other

[50] Physicists sometimes adopt this view by introducing the 'world lines' of particles as a device for reasoning about motion. We've also already mentioned (section 2.1.3) that this is the natural way to think about signal-processing systems. We will explore applications of streams to signal processing in section 3.5.3.

hand, the stream framework raises difficulties of its own, and the question of which modeling technique leads to more modular and more easily maintained systems remains open.

3.5.1 Streams Are Delayed Lists

As we saw in section 2.1.3, sequences can serve as standard interfaces for combining program modules. We formulated powerful abstractions for manipulating sequences, such as `map`, `filter`, and `accumulate`, that capture a wide variety of operations in a manner that is both succinct and elegant.

Unfortunately, if we represent sequences as lists, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations. When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

To see why this is true, let us compare two programs for computing the sum of all the prime numbers in an interval. The first program is written in standard iterative style:[51]

```
function sum_primes(a, b) {
    function iter(count, accum) {
        if (count > b) {
            return accum;
        } else {
            if (is_prime(count)) {
                return iter(count + 1, count + accum);
            } else {
                return iter(count + 1, accum);
            }
        }
    }
    return iter(a, 0);
}
```

The second program performs the same computation using the sequence operations of section 2.1.3:

```
function sum_primes(a, b) {
    return accumulate((x, y) => x + y,
                      0,
                      filter(is_prime,
                             enumerate_interval(a, b)));
}
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, the filter in the second program cannot do any testing until `enumerate_interval`

[51] Assume that we have a predicate `is_prime` (e.g., as in section ??) that tests for primality.

has constructed a complete list of the numbers in the interval. The filter generates another list, which in turn is passed to `accumulate` before being collapsed to form a sum. Such large intermediate storage is not needed by the first program, which we can think of as enumerating the interval incrementally, adding each prime to the sum as it is generated.

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

```
head(tail(filter(is_prime,
                 enumerate_interval(10000, 1000000))));
```

This expression does find the second prime when given enough time and space, but the computational overhead is outrageous. We construct a list of almost a million integers, filter this list by testing each element for primality, and then ignore almost all of the result. In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.

Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists. With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation. The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists. In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

In their most basic form, streams are similar to lists. The empty stream is `null`, a non-empty stream is a pair, and the `head` of the pair is a data item. However, the `tail` of a pair that represents a non-empty stream is not a stream, but a *nullary function that returns a stream.* The stream returned by the function, we call *the tail of the stream.* If we have a data item `x` and a stream `s`, we can construct a stream whose head is `x` and whose tail is `s` by evaluating `pair(x, () => s)`.

In order to access the data item of a non-empty stream, we just use `head` as with lists. In order to access the tail of a stream `s`, we need to *apply* `tail(s)`, i.e. evaluate `(tail(s))()`. For convenience, we therefore define

```
function stream_tail(stream) {
    return tail(stream)();
}
```

We can make and use streams, in just the same way as we can make and use lists, to represent aggregate data arranged in a sequence. In particular, we can build stream analogs of the list

operations from chapter 2, such as `list_ref`, `map`, and `for_each`:[52]

```
function stream_ref(s, n) {
    return n === 0
           ? head(s)
           : stream_ref(stream_tail(s), n - 1);
}
function stream_map(f, s) {
    return is_null(s)
           ? null
           : pair(f(head(s)),
                  () => stream_map(f, stream_tail(s)));
}
function stream_for_each(fun, s) {
    if (is_null(s)) {
        return true;
    } else {
        fun(head(s));
        return stream_for_each(fun, stream_tail(s));
    }
}
```

The function `stream_for_each` is useful for viewing streams:

```
function display_stream(s) {
    return stream_for_each(display, s);
}
```

The function that represents the tail of a stream is evaluated when it is accessed, using `stream_tail`. This design choice is reminiscent of our discussion of rational numbers in section **??**, where we saw that we can choose to implement rational numbers so that the reduction of numerator and denominator to lowest terms is performed either at construction time or at selection time. The two rational-number implementations produce the same data abstraction, but the choice has an effect on efficiency. There is a similar relationship between streams and ordinary lists. As a data abstraction, streams are the same as lists. The difference is the time at which the elements are evaluated. With ordinary lists, both the `head` `tail` are evaluated at construction time. With streams, the `tail` is evaluated at selection time.

The tail of a stream is 'wrapped' in a function. It is a *delayed expression*, a 'promise' to evaluate an expression exp at some future time. Correspondingly, `stream_tail` forces the tail to fulfill its promise. It selects the `tail` of the pair and evaluates the delayed expression found there to obtain the rest of the stream.

[52]This should bother you. The fact that we are defining such similar functions for streams and lists indicates that we are missing some underlying abstraction. Unfortunately, in order to exploit this abstraction, we will need to exert finer control over the process of evaluation than we can at present. We will discuss this point further at the end of section 3.5.4. In section 4.2, we'll develop a framework that unifies lists and streams.

Streams in action

To see how this data structure behaves, let us analyze the 'outrageous' prime computation we saw above, reformulated in terms of streams:

```
head(stream_tail(stream_filter(
                    is_prime,
                    stream_enumerate_interval(10000,
                                              1000000))));
```

We will see that it does indeed work efficiently.

We begin by calling `stream_enumerate_interval` with the arguments 10,000 and 1,000,000. The function `stream_enumerate_interval` is the stream analog of `enumerate_interval` (section 2.1.3):

```
function stream_enumerate_interval(low, high) {
    return low > high
           ? null
           : pair(low,
                  () => stream_enumerate_interval(low + 1,
                                                  high));
}
```

and thus the result returned by `stream_enumerate_interval`, formed by the `pair`, is[53]

```
pair(10000, () => stream_enumerate_interval(10001, 1000000));
```

That is, `stream_enumerate_interval` returns a stream represented as a pair whose `head` is 10,000 and whose `tail` is a promise to enumerate more of the interval if so requested. This stream is now filtered for primes, using the stream analog of the `filter` function (section 2.1.3):

```
function stream_filter(pred, s) {
    return is_null(s)
           ? null
           : pred(head(s))
             ? pair(head(s),
                    () => stream_filter(pred,
                                        stream_tail(s)))
             : stream_filter(pred,
                             stream_tail(s));
}
```

The function `stream_filter` tests the `head` of the stream (which is 10,000). Since this is not prime, `stream_filter` examines the tail of its input stream. The call to `stream_tail` forces evaluation of the delayed `stream_enumerate_interval`, which now returns

[53] The numbers shown here do not really appear in the delayed expression. What actually appears is the original expression, in an environment in which the variables are bound to the appropriate numbers. For example, `low + 1` with `low` bound to 10,000 actually appears where `10001` is shown.

```
pair(10001, () => stream_enumerate_interval(10002, 1000000));
```

The function stream_filter now looks at the head of this stream, 10,001, sees that this is not prime either, forces another stream_tail, and so on, until stream_enumerate_interval yields the prime 10,007, whereupon stream_filter, according to its definition, returns

```
pair(head(stream), stream_filter(pred, stream_tail(stream)));
```

which in this case is

```
pair(10007,
     () => stream_filter(is_prime,
               pair(10008,
                    () => stream_enumerate_interval(10009,
                                                    1000000))
               )
    );
```

This result is now passed to stream_tail in our original expression. This forces the delayed stream_filter, which in turn keeps forcing the delayed stream_enumerate_interval until it finds the next prime, which is 10,009. Finally, the result passed to head in our original expression is

```
pair(10009,
     () => stream_filter(is_prime,
               pair(10010,
                    () => stream_enumerate_interval(10011,
                                                    1000000))
               )
    );
```

The function head returns 10,009, and the computation is complete. Only as many integers were tested for primality as were necessary to find the second prime, and the interval was enumerated only as far as was necessary to feed the prime filter.

In general, we can think of delayed evaluation as ‘demand-driven’ programming, whereby each stage in the stream process is activated only enough to satisfy the next stage. What we have done is to decouple the actual order of events in the computation from the apparent structure of our functions. We write functions as if the streams existed ‘all at once’ when, in reality, the computation is performed incrementally, as in traditional programming styles.

An optimization

When we construct stream pairs, we delay the evaluation of their tail expressions by wrapping these expressions in a function. We force their evaluation when needed, by applying the function.

This implementation suffices for streams to work as advertised, but there is an important optimization that we can include. In many applications, we end up forcing the same delayed object many times. This can lead to serious inefficiency in recursive programs involving streams. (See exercise 3.57.) The solution is to build delayed objects so that the first time they are forced, they store the value that is computed. Subsequent forcings will simply return the stored value without repeating the computation. In other words, we implement the construction of stream pairs as a memoized function similar to the one described in exercise 3.27. One way to accomplish this is to use the following function, which takes as argument a function (of no arguments) and returns a memoized version of the function. The first time the memoized function is run, it saves the computed result. On subsequent evaluations, it simply returns the result.

```
function memo(fun) {
    let already_run = false;
    let result = undefined;
    return () => {
                    if (!already_run) {
                        result = fun();
                        already_run = true;
                        return result;
                    } else {
                        return result;
                    }
                };
}
```

We can make use of `memo` whenever we construct a stream pair. For example, instead of

```
function stream_map(f, s) {
    return is_null(s)
           ? null
           : pair(f(head(s)),
                  () => stream_map(f, stream_tail(s)));
}
```

we can define an optimized function `stream_map` as follows:[54]

[54]There are many possible implementations of streams other than the one described in this section. Delayed evaluation, which is the key to making streams practical, was inherent in Algol 60's *call-by-name* parameter-passing method. The use of this mechanism to implement streams was first described by Landin (1965). Delayed evaluation for streams was introduced into Lisp by Friedman and Wise (1976). In their implementation, cons always delays evaluating its arguments, so that lists automatically behave as streams. The memoizing optimization is also known as *call-by-need*. The Algol community would refer to our original delayed objects as *call-by-name*

```
function stream_map_optimized(f, s) {
    return is_null(s)
           ? null
           : pair(f(head(s)),
                  memo( () => stream_map_optimized(
                                  f, stream_tail(s)) ));
}
```

Exercise 3.50

Define a function `stream_combine` that takes a binary function and two streams as arguments and returns a stream whose elements are the results of applying the function pairwise to the corresponding elements of the argument streams.

```
function stream_combine(f, s1, s2) {
    ...
}
```

Exercise 3.51

In order to take a closer look at delayed evaluation, we will use the following function, which simply returns its argument after printing it:

```
function show(x) {
    display(x);
    return x;
}
```

What does the interpreter print in response to evaluating each expression in the following sequence?[55]

```
let x = stream_map(show, stream_enumerate_interval(0, 10));
stream_ref(x, 5);
stream_ref(x, 7);
```

What does the evaluator print if `stream_map_optimized` is used instead of `stream_map`?

thunks and to the optimized versions as *call-by-need thunks*.

[55] Exercises such as 3.51 and 3.52 are valuable for testing our understanding of how delayed evaluation works. On the other hand, intermixing delayed evaluation with printing—and, even worse, with assignment—is extremely confusing, and instructors of courses on computer languages have traditionally tormented their students with examination questions such as the ones in this section. Needless to say, writing programs that depend on such subtleties is odious programming style. Part of the power of stream processing is that it lets us ignore the order in which events actually happen in our programs. Unfortunately, this is precisely what we cannot afford to do in the presence of assignment, which forces us to be concerned with time and change.

```
let x = stream_map_optimized(
            show,
            stream_enumerate_interval(0, 10));
stream_ref(x, 5);
stream_ref(x, 7);
```

Exercise 3.52

Consider the sequence of expressions

```
let sum = 0;

function accum(x) {
    sum = x + sum;
    return sum;
}

const seq = stream_map(
                accum,
                stream_enumerate_interval(1, 20));
const y = stream_filter(is_even, seq);

const z = stream_filter(x => x % 5 === 0, seq);

stream_ref(y, 7);

display_stream(z);
```

What is the value of `sum` after each of the above expressions is evaluated? What is the printed response to evaluating the `stream_ref` and `display_stream` expressions? Would these responses differ if we had applied the function `memo` on every tail of every constructed stream pair, as suggested in the optimization above? Explain.

3.5.2 Infinite Streams

We have seen how to support the illusion of manipulating streams as complete entities even though, in actuality, we compute only as much of the stream as we need to access. We can exploit this technique to represent sequences efficiently as streams, even if the sequences are very long. What is more striking, we can use streams to represent sequences that are infinitely long. For instance, consider the following definition of the stream of positive integers:

```
function integers_starting_from(n) {
    return pair(n,
                () => integers_starting_from(n + 1)
               );
}
```

This makes sense because `integers` will be a pair whose `head` is 1 and whose `tail` is a promise to produce the integers beginning with 2. This is an infinitely long stream, but in any given time we can examine only a finite portion of it. Thus, our programs will never know that the entire infinite stream is not there.

Using `integers` we can define other infinite streams, such as the stream of integers that are not divisible by 7:

```
function is_divisible(x, y) {
    return x % y === 0;
}

const no_sevens =
    stream_filter(x => ! is_divisible(x, 7),
                  integers);
```

Then we can find integers not divisible by 7 simply by accessing elements of this stream:

```
stream_ref(no_sevens, 100);
```

In analogy with `integers`, we can define the infinite stream of Fibonacci numbers:

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}

const fibs = fibgen(0, 1);
```

The function `fibs` is a pair whose `head` is 0 and whose `tail` is a promise to evaluate `fibgen(1, 1)`. When we evaluate this delayed `fibgen(1, 1)`, it will produce a pair whose `head` is 1 and whose `tail` is a promise to evaluate `fibgen(1, 2)`, and so on.

For a look at a more exciting infinite stream, we can generalize the `no_sevens` example to construct the infinite stream of prime numbers, using a method known as the *sieve of Eratosthenes*.[56]

[56] Eratosthenes, a third-century B.C. Alexandrian Greek philosopher, is famous for giving the first accurate

We start with the integers beginning with 2, which is the first prime. To get the rest of the primes, we start by filtering the multiples of 2 from the rest of the integers. This leaves a stream beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of this stream. This leaves a stream beginning with 5, which is the next prime, and so on. In other words, we construct the primes by a sieving process, described as follows: To sieve a stream `S`, form a stream whose first element is the first element of `S` and the rest of which is obtained by filtering all multiples of the first element of `S` out of the rest of `S` and sieving the result. This process is readily described in terms of stream operations:

```
function sieve(stream) {
    return pair(head(stream),
                () => sieve(stream_filter(
                                x => !is_divisible(x,
                                          head(stream)),
                                stream_tail(stream)
                               )
                            )
                );
}
```

Now to find a particular prime we need only ask for it:

```
stream_ref(primes, 50);
```

It is interesting to contemplate the signal-processing system set up by `sieve`, shown in the 'Henderson diagram' in Figure 3.31.[57] The input stream feeds into an 'unpairer' that separates the first element of the stream from the rest of the stream. The first element is used to construct a divisibility filter, through which the rest is passed, and the output of the filter is fed to another sieve box. Then the original first element is `pair`ed onto the output of the internal sieve to form the output stream. Thus, not only is the stream infinite, but the signal processor is also infinite, because the sieve contains a sieve within it.

estimate of the circumference of the Earth, which he computed by observing shadows cast at noon on the day of the summer solstice. Eratosthenes's sieve method, although ancient, has formed the basis for special-purpose hardware 'sieves' that, until the 1970s, were the most powerful tools in existence for locating large primes. Since then, however, these methods have been superseded by outgrowths of the probabilistic techniques discussed in section **??**.

[57] We have named these figures after Peter Henderson, who was the first person to show us diagrams of this sort as a way of thinking about stream processing. Each solid line represents a stream of values being transmitted. The dashed line from the `head` to the `pair` and the `filter` indicates that this is a single value rather than a stream.

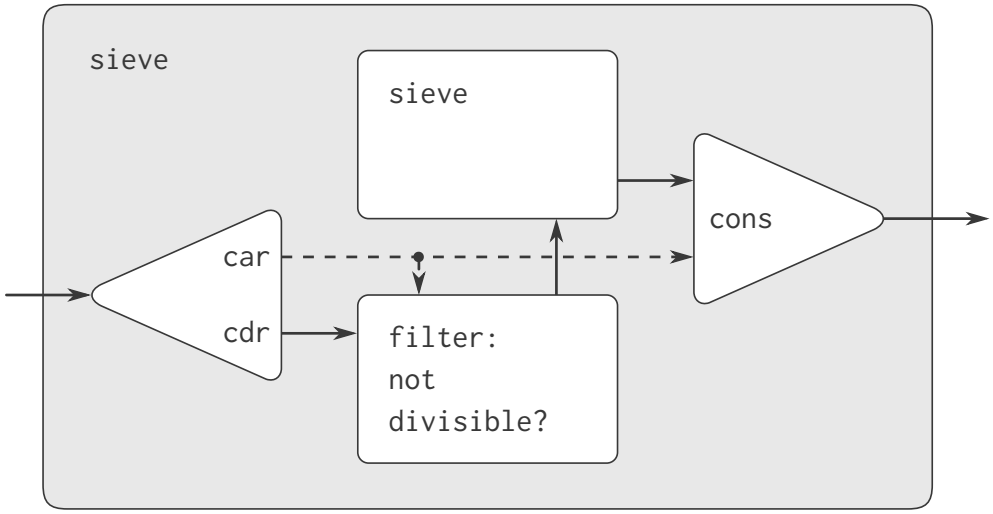


Figure 3.31: The prime sieve viewed as a signal-processing system.

Defining streams implicitly

The `integers` and `fibs` streams above were defined by specifying 'generating' functions that explicitly compute the stream elements one by one. An alternative way to specify streams is to take advantage of delayed evaluation to define streams implicitly. For example, the following expression defines the stream `ones` to be an infinite stream of ones:

```
const ones = pair(1, () => ones);
```

This works much like the definition of a recursive function: `ones` is a pair whose `head` is 1 and whose `tail` is a promise to evaluate `ones`. Evaluating the `tail` gives us again a 1 and a promise to evaluate `ones`, and so on.

We can do more interesting things by manipulating streams with operations such as `add_streams`, which produces the elementwise sum of two given streams:[58]

```
function add_streams(s1, s2) {
    return stream_combine((x1, x2) => x1 + x2, s1, s2);
}
```

Now we can define the integers as follows:

```
const integers = pair(1, () => add_streams(ones, integers));
```

This defines `integers` to be a stream whose first element is 1 and the rest of which is the sum of `ones` and `integers`. Thus, the second element of `integers` is 1 plus the first element of `integers`, or 2; the third element of `integers` is 1 plus the second element of `integers`, or 3; and so on. This definition works because, at any point, enough of the `integers` stream has been generated so that we can feed it back into the definition to produce the next integer.

[58] This uses the function `stream_merge` from exercise 3.50.

We can define the Fibonacci numbers in the same style:

```
const fibs = pair(0,
                  () => pair(1,
                             () => add_streams(stream_tail(
                                                 fibs))
                            )
                 );
```

This definition says that `fibs` is a stream beginning with 0 and 1, such that the rest of the stream can be generated by adding `fibs` to itself shifted by one place:

```
        1   1   2   3   5   8   13  21  ... = stream_tail(fibs)
        0   1   1   2   3   5   8   13  ... = fibs
-------------------------------------------------------------
0   1   1   2   3   5   8   13  21  34  ... = fibs
```

The function `scale_stream` is also useful in formulating such stream definitions. This multiplies each item in a stream by a given constant:

```
function scale_stream(stream, factor) {
    return stream_map(x => x * factor,
                      stream);
}
```

For example,

```
const double = pair(1, () => scale_stream(double, 2));
```

produces the stream of powers of 2: 1, 2, 4, 8, 16, 32, ...

An alternate definition of the stream of primes can be given by starting with the integers and filtering them by testing for primality. We will need the first prime, 2, to get started:

```
const primes = pair(2,
                    () => stream_filter(
                              is_prime,
                              integers_starting_from(3))
                   );
```

This definition is not so straightforward as it appears, because we will test whether a number n is prime by checking whether n is divisible by a prime (not by just any integer) less than or equal to $\sqrt{n}$:

```
function is_prime(n) {
    function iter(ps) {
        return square(head(ps)) > n
               ? true
               : is_divisible(n, head(ps))
                 ? false
                 : iter(stream_tail(ps));
    }
```

```
    return iter(primes);
}
```

This is a recursive definition, since `primes` is defined in terms of the `is_prime` predicate, which itself uses the `primes` stream. The reason this function works is that, at any point, enough of the `primes` stream has been generated to test the primality of the numbers we need to check next. That is, for every n we test for primality, either n is not prime (in which case there is a prime already generated that divides it) or n is prime (in which case there is a prime already generated—i.e., a prime less than n—that is greater than $\sqrt{n}$).[59]

Exercise 3.53

Without running the program, describe the elements of the stream defined by

```
const s = pair(1, () => add_streams(s, s));
```

Exercise 3.54

Define a function `mul_streams`, analogous to `add_streams`, that produces the elementwise product of its two input streams. Use this together with the stream of `integers` to complete the following definition of the stream whose nth element (counting from 0) is $n + 1$ factorial:

```
// mul_streams to be written by students
const factorials = pair(1, () => mul_streams(???, ???));
```

Exercise 3.55

Define a function `partial_sums` that takes as argument a stream S and returns the stream whose elements are $S_0, S_0 + S_1, S_0 + S_1 + S_2, \ldots$. For example, `partial_sums(integers)` should be the stream $1, 3, 6, 10, 15, \ldots$.

Exercise 3.56

A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them

[59] This last point is very subtle and relies on the fact that $p_{n+1} \leq p_n^2$. (Here, p_k denotes the kth prime.) Estimates such as these are very difficult to establish. The ancient proof by Euclid that there are an infinite number of primes shows that $p_{n+1} \leq p_1 p_2 \cdots p_n + 1$, and no substantially better result was proved until 1851, when the Russian mathematician P. L. Chebyshev established that $p_{n+1} \leq 2p_n$ for all n. This result, originally conjectured in 1845, is known as *Bertrand's hypothesis*. A proof can be found in section 22.3 of Hardy and Wright 1960.

fit the requirement. As an alternative, let us call the required stream of numbers `S` and notice the following facts about it.

- `S` begins with 1.
- The elements of `scale_stream(S, 2)` are also elements of `S`.
- The same is true for `scale_stream(S, 3)` and `scale_stream(5, S)`.
- These are all the elements of `S`.

Now all we have to do is combine elements from these sources. For this we define a function `merge` that combines two ordered streams into one ordered result stream, eliminating repetitions:

```
function merge(s1, s2) {
    if (is_null(s1)) {
        return s2;
    } else if (is_null(s2)) {
        return s1;
    } else {
        const s1head = head(s1);
        const s2head = head(s2);
        if (s1head < s2head) {
            return pair(s1head,
                        () => merge(stream_tail(s1), s2)
                       );
        } else if (s1head > s2head) {
            return pair(s2head,
                        () => merge(s1, stream_tail(s2))
                       );
        } else {
            return merge(stream_tail(s1), stream_tail(s2));
        }
    }
}
```

Then the required stream may be constructed with `merge`, as follows:

```
const S = pair(1, () => merge(??, ??));
```

Fill in the missing expressions in the places marked ?? above.

Exercise 3.57

How many additions are performed when we compute the *n*th Fibonacci number using the definition of `fibs` based on the `add_streams` function, implemented using `pair(..., () => ...)`

as described in the beginning of section 3.5.1? Show that the number of additions is exponentially greater than if we had implemented `add_streams` using the optimization using `pair(..., memo( () => ... ))` described in the last part of section 3.5.1.[60]

Exercise 3.58

Give an interpretation of the stream computed by the function :

```
function expand(num, den, radix) {
    return pair(quotient(num * radix, den),
                expand((num * radix) % den, den, radix));
}
```

where the function `quotient` computes integer division, in which the fractional part (remainder) is discarded. What are the successive elements produced by `expand(1, 7, 10)` ? What is produced by `expand(3, 8, 10)` ?

Exercise 3.59

In section **??** we saw how to implement a polynomial arithmetic system representing polynomials as lists of terms. In a similar way, we can work with *power series*, such as

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \cdots ,$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \cdots ,$$

$$\sin x = x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \cdots ,$$

represented as infinite streams. We will represent the series $a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$ as the stream whose elements are the coefficients $a_0, a_1, a_2, a_3, \ldots$.

a. The integral of the series $a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$ is the series

$$c + a_0 x + \frac{1}{2} a_1 x^2 + \frac{1}{3} a_2 x^3 + \frac{1}{4} a_3 x^4 + \cdots$$

where c is any constant. Define a function `integrate_series` that takes as input a stream $a_0, a_1, a_2, \ldots$ representing a power series and returns the stream $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \ldots$ of coefficients of the non-constant terms of the integral of the series. (Since the result has

[60] This exercise shows how call-by-need is closely related to ordinary memoization as described in exercise 3.27. In that exercise, we used assignment to explicitly construct a local table. Our call-by-need stream optimization effectively constructs such a table automatically, storing values in the previously forced parts of the stream.

no constant term, it doesn't represent a power series; when we use `integrate-series`, we will `pair` with the appropriate constant.)

b. The function $x \mapsto e^x$ is its own derivative. This implies that e^x and the integral of e^x are the same series, except for the constant term, which is $e^0 = 1$. Accordingly, we can generate the series for e^x as

```
const exp_series =
    pair(1, () => integrate_series(exp_series));
```

Show how to generate the series for sine and cosine, starting from the facts that the derivative of sine is cosine and the derivative of cosine is the negative of sine:

```
const cosine_series = pair(1, ??);
const sine_series = pair(0, ??);
```

Exercise 3.60

With power series represented as streams of coefficients as in exercise 3.59, adding series is implemented by `add-streams`. Complete the definition of the following function for multiplying series:

```
function mul_series(s1, s2) {
    pair(??, add_streams(??, ??));
}
```

You can test your function by verifying that $sin^2x+cos^2x = 1$, using the series from exercise 3.59.

Exercise 3.61

Let S be a power series (exercise 3.59) whose constant term is 1. Suppose we want to find the power series $1/S$, that is, the series X such that $S \cdot X = 1$. Write $S = 1 + S_R$ where S_R is the part of S after the constant term. Then we can solve for X as follows:

$$\begin{aligned} S \cdot X &= 1 \\ (1 + S_R) \cdot X &= 1 \\ X + S_R \cdot X &= 1 \\ X &= 1 - S_R \cdot X \end{aligned}$$

In other words, X is the power series whose constant term is 1 and whose higher-order terms are given by the negative of S_R times X. Use this idea to write a function `invert_unit_series` that computes $1/S$ for a power series S with constant term 1. You will need to use `mul_series` from exercise 3.60.

Exercise 3.62

Use the results of exercises 3.60 and 3.61 to define a function `div_series` that divides two power series. The function `div_series` should work for any two series, provided that the denominator series begins with a nonzero constant term. (If the denominator has a zero constant term, then `div_series` should signal an error.) Show how to use `div_series` together with the result of exercise 3.59 to generate the power series for tangent.

3.5.3 Exploiting the Stream Paradigm

Note: this section is a work in progress!

Streams with delayed evaluation can be a powerful modeling tool, providing many of the benefits of local state and assignment. Moreover, they avoid some of the theoretical tangles that accompany the introduction of assignment into a programming language.

The stream approach can be illuminating because it allows us to build systems with different module boundaries than systems organized around assignment to state variables. For example, we can think of an entire time series (or signal) as a focus of interest, rather than the values of the state variables at individual moments. This makes it convenient to combine and compare components of state from different moments.

Formulating iterations as stream processes

In section **??**, we introduced iterative processes, which proceed by updating state variables. We know now that we can represent state as a 'timeless' stream of values rather than as a set of variables to be updated. Let's adopt this perspective in revisiting the square-root function from section **??**. Recall that the idea is to generate a sequence of better and better guesses for the square root of x by applying over and over again the function that improves guesses:

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

In our original `sqrt` function, we made these guesses be the successive values of a state variable. Instead we can generate the infinite stream of guesses, starting with an initial guess of 1:[61]

```
function sqrt_stream(x) {
    const guesses =
        pair(1.0,
             () => stream_map(guess => sqrt_improve(guess, x),
```

[61] We can't use **let** to bind the local variable guesses, because the value of guesses depends on guesses itself. Exercise 3.63 addresses why we want a local variable here.

```
                                guesses);
            );
    return guesses;
}
display(eval_stream(sqrt_stream(2), 5));
// [1, [1.5, [1.4166666666666665, [1.4142156862745097,
// [1.4142135623746899, null]]]]]
```

We can generate more and more terms of the stream to get better and better guesses. If we like, we can write a function that keeps generating terms until the answer is good enough. (See exercise 3.64.)

Another iteration that we can treat in the same way is to generate an approximation to π, based upon the alternating series that we saw in section **??**:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

We first generate the stream of summands of the series (the reciprocals of the odd integers, with alternating signs). Then we take the stream of sums of more and more terms (using the `partial_sums` function of exercise 3.55) and scale the result by 4:

```
function pi_summands(n) {
    return pair(1.0 / n,
                () => stream_map(x => -x,
                                 pi_summands(n + 2))
               );
}

const pi_stream =
    scale_stream(partial_sums(pi_summands(1)), 4);
display_stream(eval_stream(pi_stream, 8));
// [4, [2.666666666666667, [3.466666666666667,
// [2.8952380952380956, [3.3396825396825403,
// [2.9760461760461765, [3.2837384837384844,
// [3.017071817071818, null]]]]]]]]
```

This gives us a stream of better and better approximations to π, although the approximations converge rather slowly. Eight terms of the sequence bound the value of π between 3.284 and 3.017.

So far, our use of the stream of states approach is not much different from updating state variables. But streams give us an opportunity to do some interesting tricks. For example, we can transform a stream with a *sequence accelerator* that converts a sequence of approximations to a new sequence that converges to the same value as the original, only faster.

One such accelerator, due to the eighteenth-century Swiss mathematician Leonhard Euler,

works well with sequences that are partial sums of alternating series (series of terms with alternating signs). In Euler's technique, if S_n is the nth term of the original sum sequence, then the accelerated sequence has terms

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Thus, if the original sequence is represented as a stream of values, the transformed sequence is given by

```
function euler_transform(s) {
    const s0 = stream_ref(s, 0);
    const s1 = stream_ref(s, 1);
    const s2 = stream_ref(s, 2);
    return pair(s2 -
                square(s2 - s1) / (s0 + (-2) * s1 + s2),
                () => euler_transform(stream_tail(s)));
}
```

We can demonstrate Euler acceleration with our sequence of approximations to π:

```
display_stream(euler_transform(pi_stream));
// 3.166666666666667
// 3.1333333333333337
// 3.1452380952380956
// 3.13968253968254
// 3.1427128427128435
// 3.1408813408813416
// 3.142071817071818
// 3.1412548236077655
// ...
```

Even better, we can accelerate the accelerated sequence, and recursively accelerate that, and so on. Namely, we create a stream of streams (a structure we'll call a *tableau*) in which each stream is the transform of the preceding one:

```
function make_tableau(transform, s) {
    return pair(s, () => make_tableau(transform, transform(s)));
}
```

The tableau has the form

$$\begin{array}{cccccc} s_{00} & s_{01} & s_{02} & s_{03} & s_{04} & \ldots \\ & s_{10} & s_{11} & s_{12} & s_{13} & \ldots \\ & & s_{20} & s_{21} & s_{22} & \ldots \\ & & & & \ldots & \end{array}$$

Finally, we form a sequence by taking the first term in each row of the tableau:

```
function accelerated_sequence(transform, s) {
```

```
    return stream_map(head, make_tableau(transform, s));
}
```

We can demonstrate this kind of 'super-acceleration' of the π sequence:

```
display(eval_stream(accelerated_sequence(euler_transform,
                                         pi_stream),
                    8));
// [4, [3.166666666666667, [3.142105263157895,
// [3.141599357319005, [3.1415927140337785, [3.1415926539752927,
// [3.1415926535911765, [3.141592653589778, null]]]]]]]]
```

The result is impressive. Taking eight terms of the sequence yields the correct value of π to 14 decimal places. If we had used only the original π sequence, we would need to compute on the order of 10^{13} terms (i.e., expanding the series far enough so that the individual terms are less then 10^{-13}) to get that much accuracy!

We could have implemented these acceleration techniques without using streams. But the stream formulation is particularly elegant and convenient because the entire sequence of states is available to us as a data structure that can be manipulated with a uniform set of operations.

Exercise 3.63

Louis Reasoner asks why the `sqrt_stream` function was not written in the following more straightforward way, without the local variable guesses:

```
function sqrt_stream(x) {
    return pair(1.0,
                () => stream_map(guess =>
                                   sqrt_improve(guess, x),
                                 sqrt_stream(x))
               );
}
```

Alyssa P. Hacker replies that this version of the function is considerably less efficient because it performs redundant computation. Explain Alyssa's answer. Would the two versions still differ in efficiency if our implementation of `delay` used only `???` without using the optimization provided by `memo-proc` (section 3.5.1)?

Exercise 3.64

Write a function `stream_limit` that takes as arguments a stream and a number (the tolerance). It should examine the stream until it finds two successive elements that differ in absolute value by less than the tolerance, and return the second of the two elements. Using this, we could

compute square roots up to a given tolerance by

```
function sqrt(x, tolerance) {
    return stream_limit(sqrt_stream(x), tolerance);
}
```

Exercise 3.65

Use the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$$

to compute three sequences of approximations to the natural logarithm of 2, in the same way we did above for π. How rapidly do these sequences converge?

Infinite streams of pairs

In section 2.1.3, we saw how the sequence paradigm handles traditional nested loops as processes defined on sequences of pairs. If we generalize this technique to infinite streams, then we can write programs that are not easily represented as loops, because the 'looping' must range over an infinite set.

For example, suppose we want to generalize the `prime_sum_pairs` function of section 2.1.3 to produce the stream of pairs of *all* integers (i, j) with $i \leq j$ such that $i + j$ is prime. If `int_pairs` is the sequence of all pairs of integers (i, j) with $i \leq j$, then our required stream is simply[62]

```
stream_filter(pair => is_prime(head(pair) + head(tail(pair))),
              int_pairs);
```

Our problem, then, is to produce the stream `int_pairs`. More generally, suppose we have two streams $S = (S_i)$ and $T = (T_j)$, and imagine the infinite rectangular array

$$\begin{array}{cccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ (S_1, T_0) & (S_1, T_1) & (S_1, T_2) & \ldots \\ (S_2, T_0) & (S_2, T_1) & (S_2, T_2) & \ldots \\ \ldots & & & \end{array}$$

We wish to generate a stream that contains all the pairs in the array that lie on or above the diagonal, i.e., the pairs

$$\begin{array}{cccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ & (S_1, T_1) & (S_1, T_2) & \ldots \\ & & (S_2, T_2) & \ldots \\ & & & \ldots \end{array}$$

[62] As in section 2.1.3, we represent a pair of integers as a list rather than a JavaScript pair.

(If we take both S and T to be the stream of integers, then this will be our desired stream `int_pairs`.)

Call the general stream of pairs `pairs(S, T)`, and consider it to be composed of three parts: the pair (S_0, T_0), the rest of the pairs in the first row, and the remaining pairs:[63]

(S_0, T_0)	(S_0, T_1)	(S_0, T_2)	...
	(S_1, T_1)	(S_1, T_2)	...
		(S_2, T_2)	...
			...

Observe that the third piece in this decomposition (pairs that are not in the first row) is (recursively) the pairs formed from `stream_tail(S)` and `stream_tail(T)`. Also note that the second piece (the rest of the first row) is

```
stream_map(x => list(head(s), x),
           stream_tail(t));
```

Thus we can form our stream of pairs as follows:

```
function pairs(s, t) {
    return pair(list(head(s), head(t)),
                () => <combine_in_some_way>(
                          stream_map(x => list(head(s), x),
                                     stream_tail(t)),
                          pairs(stream_tail(s), stream_tail(t)))
               );
}
```

In order to complete the function, we must choose some way to combine the two inner streams. One idea is to use the stream analog of the append function from section 2.1.1:

```
function stream_append(s1, s2) {
    return is_null(s1)
        ? s2
        : pair(head(s1),
               () => stream_append(stream_tail(s1), s2)
              );
}
```

This is unsuitable for infinite streams, however, because it takes all the elements from the first stream before incorporating the second stream. In particular, if we try to generate all pairs of positive integers using

```
pairs(integers, integers);
```

[63] See exercise 3.68 for some insight into why we chose this decomposition.

our stream of results will first try to run through all pairs with the first integer equal to 1, and hence will never produce pairs with any other value of the first integer.

To handle infinite streams, we need to devise an order of combination that ensures that every element will eventually be reached if we let our program run long enough. An elegant way to accomplish this is with the following `interleave` function:[64]

```
function interleave(s1, s2) {
    return is_null(s1)
        ? s2;
        : pair(head(s1),
               () => interleave(s2, stream_tail(s1))
              );
}
```

Since `interleave` takes elements alternately from the two streams, every element of the second stream will eventually find its way into the interleaved stream, even if the first stream is infinite.

We can thus generate the required stream of pairs as

```
function pairs(s, t) {
    return pair(list(head(s), head(t)),
                () => interleave(stream_map(x => list(head(s),
                                                      x),
                                            stream_tail(t)),
                                 pairs(stream_tail(s),
                                       stream_tail(t))));
}
```

Exercise 3.66

Examine the stream `pairs(integers, integers)`. Can you make any general comments about the order in which the pairs are placed into the stream? For example, about how many pairs precede the pair (1,100)? the pair (99,100)? the pair (100,100)? (If you can make precise mathematical statements here, all the better. But feel free to give more qualitative answers if you find yourself getting bogged down.)

Exercise 3.67

Modify the `pairs` function so that `pairs(integers, integers)` will produce the stream of *all* pairs of integers (i, j) (without the condition $i \leq j$). Hint: You will need to mix in an additional stream.

[64] The precise statement of the required property on the order of combination is as follows: There should be a function f of two arguments such that the pair corresponding to element i of the first stream and element j of the second stream will appear as element number $f(i, j)$ of the output stream. The trick of using `interleave` to accomplish this was shown to us by David Turner, who employed it in the language KRC (Turner 1981).

Exercise 3.68

Louis Reasoner thinks that building a stream of pairs from three parts is unnecessarily complicated. Instead of separating the pair (S_0, T_0) from the rest of the pairs in the first row, he proposes to work with the whole first row, as follows:

```
function pairs(s, t) {
    return interleave(stream_map(x => list(head(s), x),
                                 t),
                      pair(stream_tail(s), stream_tail(t)));
}
```

Does this work? Consider what happens if we evaluate `pairs(integers, integers)` using Louis's definition of `pairs`.

Exercise 3.69

Write a function `triples` that takes three infinite streams, S, T, and U, and produces the stream of triples (S_i, T_j, U_k) such that $i \leq j \leq k$. Use `triples` to generate the stream of all Pythagorean triples of positive integers, i.e., the triples (i, j, k) such that $i \leq j$ and $i^2 + j^2 = k^2$.

Exercise 3.70

It would be nice to be able to generate streams in which the pairs appear in some useful order, rather than in the order that results from an *ad hoc* interleaving process. We can use a technique similar to the `merge` function of exercise 3.56, if we define a way to say that one pair of integers is 'less than' another. One way to do this is to define a 'weighting function' $W(i, j)$ and stipulate that (i_1, j_1) is less than (i_2, j_2) if $W(i_1, j_1) < W(i_2, j_2)$. Write a function `merge_weighted` that is like `merge`, except that `merge_weighted` takes an additional argument `weight`, which is a function that computes the weight of a pair, and is used to determine the order in which elements should appear in the resulting merged stream.[65] Using this, generalize `pairs` to a function `weighted_pairs` that takes two streams, together with a function that computes a weighting function, and generates the stream of pairs, ordered according to weight. Use your function to generate

a. the stream of all pairs of positive integers (i, j) with $i \leq j$ ordered according to the sum $i + j$

b. the stream of all pairs of positive integers (i, j) with $i \leq j$, where neither i nor j is divisible by 2, 3, or 5, and the pairs are ordered according to the sum $2i + 3j + 5ij$.

[65] We will require that the weighting function be such that the weight of a pair increases as we move out along a row or down along a column of the array of pairs.

Exercise 3.71

Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called *Ramanujan numbers*, in honor of the mathematician Srinivasa Ramanujan.[66] Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$ (see exercise 3.70), then search the stream for two consecutive pairs with the same weight. Write a function to generate the Ramanujan numbers. The first such number is 1,729. What are the next five?

Exercise 3.72

In a similar way to exercise 3.71 generate a stream of all numbers that can be written as the sum of two squares in three different ways (showing how they can be so written).

Streams as signals

We began our discussion of streams by describing them as computational analogs of the 'signals' in signal-processing systems. In fact, we can use streams to model signal-processing systems in a very direct way, representing the values of a signal at successive time intervals as consecutive elements of a stream. For instance, we can implement an *integrator* or *summer* that, for an input stream $x = (x_i)$, an initial value C, and a small increment dt, accumulates the sum

$$S_i = C + \sum_{j=1}^{i} x_j \, dt$$

and returns the stream of values $S = (S_i)$. The following `integral` function is reminiscent of the 'implicit style' definition of the stream of integers (section 3.5.2):

```
function integral(integrand, initial_value, dt) {
    const integ = pair(initial_value,
        () => add_streams(scale_stream(integrand, dt),
                          integ);
        );
    return integ;
}
```

[66] To quote from G. H. Hardy's obituary of Ramanujan (Hardy 1921): 'It was Mr. Littlewood (I believe) who remarked that "every positive integer was one of his friends." I remember once going to see him when he was lying ill at Putney. I had ridden in taxi-cab No. 1729, and remarked that the number seemed to me a rather dull one, and that I hoped it was not an unfavorable omen. "No," he replied, "it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways."' The trick of using weighted pairs to generate the Ramanujan numbers was shown to us by Charles Leiserson.

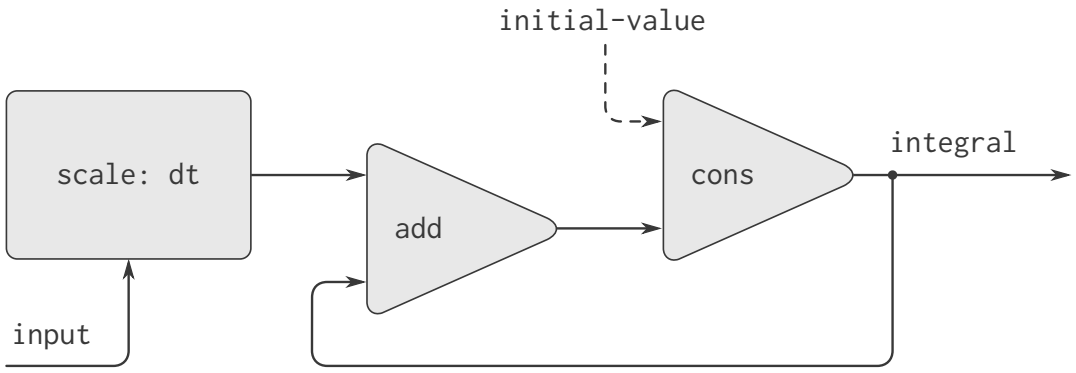


Figure 3.32: The `integral` function viewed as a signal-processing system.

Figure 3.32 is a picture of a signal-processing system that corresponds to the `integral` function. The input stream is scaled by dt and passed through an adder, whose output is passed back through the same adder. The self-reference in the definition of `int` is reflected in the figure by the feedback loop that connects the output of the adder to one of the inputs.

Exercise 3.73

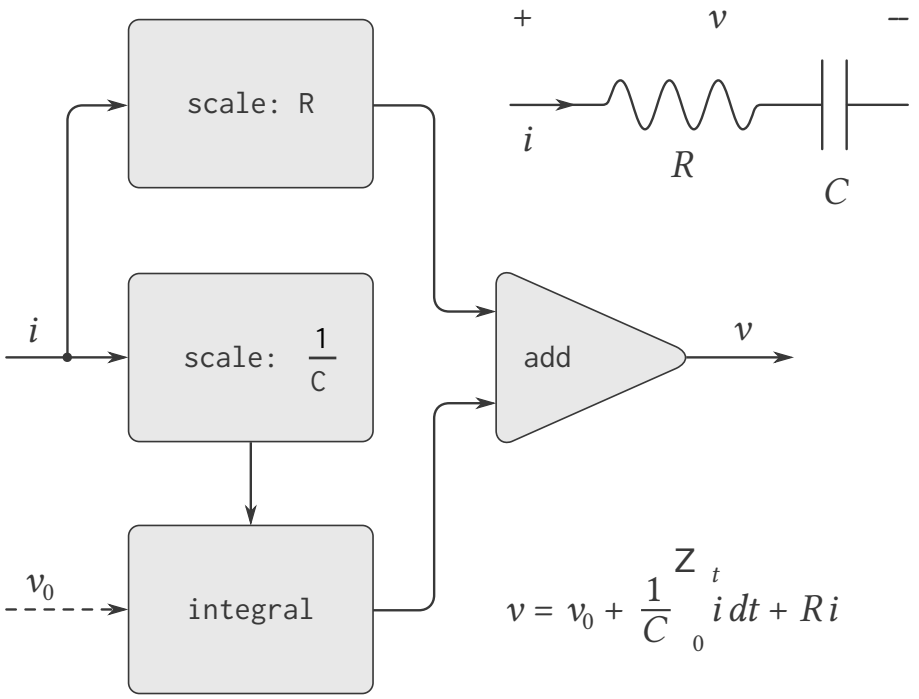


Figure 3.33: An RC circuit and the associated signal-flow diagram.

We can model electrical circuits using streams to represent the values of currents or voltages at a sequence of times. For instance, suppose we have an *RC circuit* consisting of a resistor of resistance R and a capacitor of capacitance C in series. The voltage response v of the circuit to an injected current i is determined by the formula in Figure 3.33, whose structure is shown

by the accompanying signal-flow diagram.

Write a function `RC` that models this circuit. `RC` should take as inputs the values of R, C, and dt and should return a function that takes as inputs a stream representing the current i and an initial value for the capacitor voltage v_0 and produces as output the stream of voltages v. For example, you should be able to use `RC` to model an RC circuit with $R = 5$ ohms, $C = 1$ farad, and a 0.5-second time step by evaluating **`const`** `RC1 = RC(5, 1, 0.5)`. This defines `RC1` as a function that takes a stream representing the time sequence of currents and an initial capacitor voltage and produces the output stream of voltages.

Exercise 3.74

Alyssa P. Hacker is designing a system to process signals coming from physical sensors. One important feature she wishes to produce is a signal that describes the *zero crossings* of the input signal. That is, the resulting signal should be +1 whenever the input signal changes from negative to positive, −1 whenever the input signal changes from positive to negative, and 0 otherwise. (Assume that the sign of a 0 input is positive.) For example, a typical input signal with its associated zero-crossing signal would be

```
... 1  2  1.5  1  0.5  -0.1  -2  -3  -2  -0.5  0.2  3  4 ...
... 0  0    0  0    0    -1  0   0   0     0    1  0  0 ...
```

In Alyssa's system, the signal from the sensor is represented as a stream `sense_data` and the stream `zero_crossings` is the corresponding stream of zero crossings. Alyssa first writes a function `sign_change_detector` that takes two values as arguments and compares the signs of the values to produce an appropriate 0, 1, or −1. She then constructs her zero-crossing stream as follows:

```
function make_zero_crossings(input_stream, last_value) {
    return pair(sign_change_detector(head(input_stream),
                                     last_value),
                () => make_zero_crossings(
                          stream_tail(input_stream),
                          head(input_stream)));
}
const zero_crossings = make_zero_crossings(sense_data, 0);
```

Alyssa's boss, Eva Lu Ator, walks by and suggests that this program is approximately equivalent to the following one, which uses the function `combine_streams` from exercise 3.50:

```
const zero_crossing = combine_streams(sign_change_detector,
                                      sense_data,
                                      <expression>);
```

Complete the program by supplying the indicated `<expression>`.

Exercise 3.75

Unfortunately, Alyssa's zero-crossing detector in exercise 3.74 proves to be insufficient, because the noisy signal from the sensor leads to spurious zero crossings. Lem E. Tweakit, a hardware specialist, suggests that Alyssa smooth the signal to filter out the noise before extracting the zero crossings. Alyssa takes his advice and decides to extract the zero crossings from the signal constructed by averaging each value of the sense data with the previous value. She explains the problem to her assistant, Louis Reasoner, who attempts to implement the idea, altering Alyssa's program as follows:

```
function make_zero_crossings(input_stream, last_value) {
    const avpt = (head(input_stream) + last_value) / 2;
    return pair(sign_change_detector(avpt, last_value),
        () => make_zero_crossings(
                  stream_tail(input_stream),
                  avpt);
                 );
}
```

This does not correctly implement Alyssa's plan. Find the bug that Louis has installed and fix it without changing the structure of the program. (Hint: You will need to increase the number of arguments to `make_zero_crossings`.)

Exercise 3.76

Eva Lu Ator has a criticism of Louis's approach in exercise 3.75. The program he wrote is not modular, because it intermixes the operation of smoothing with the zero-crossing extraction. For example, the extractor should not have to be changed if Alyssa finds a better way to condition her input signal. Help Louis by writing a function `smooth` that takes a stream as input and produces a stream in which each element is the average of two successive input stream elements. Then use `smooth` as a component to implement the zero-crossing detector in a more modular style.

3.5.4 Streams and Delayed Evaluation

Note: this section is a work in progress!

The `integral` function at the end of the preceding section shows how we can use streams to model signal-processing systems that contain feedback loops. The feedback loop for the adder shown in figure 3.32 is modeled by the fact that `integral`'s internal stream `int` is defined in terms of itself:

```
const integ = pair(initial_value,
    () => add_streams(scale_stream(integrand, dt),
                      integ);
        );
```

The interpreter's ability to deal with such an implicit definition depends on the delay resulting from wrapping the call of `add_streams` into a function definition. Without this delay, the interpreter could not construct `integ` before evaluating both arguments to `pair`, which would require that `integ` already be defined. In general, such a delay is crucial for using streams to model signal-processing systems that contain loops. Without a delay, our models would have to be formulated so that the inputs to any signal-processing component would be fully evaluated before the output could be produced. This would outlaw loops.

Unfortunately, stream models of systems with loops may require uses of a delay beyond the stream programming pattern seen so far. For instance, figure 3.34 shows a signal-processing system for solving the differential equation $dy/dt = f(y)$ where f is a given function. The figure shows a mapping component, which applies f to its input signal, linked in a feedback loop to an integrator in a manner very similar to that of the analog computer circuits that are actually used to solve such equations.

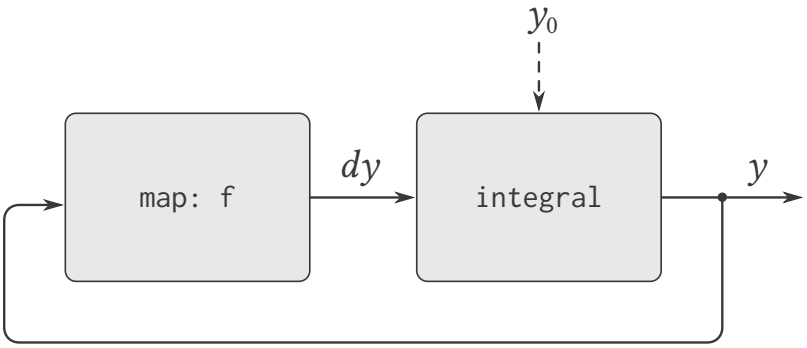


Figure 3.34: An 'analog computer circuit' that solves the equation

Assuming we are given an initial value y_0 for y, we could try to model this system using the function

```
function solve(f, y0, dt) {
    const y = integral(dy, y0, dt);
    const dy = stream_map(f, y);
    return y;
}
```

This function does not work, because in the first line of `solve` the call to `integral` requires that the input `dy` be defined, which does not happen until the second line of `solve`.

On the other hand, the intent of our definition does make sense, because we can, in principle, begin to generate the `y` stream without knowing `dy`. Indeed, `integral` and many other stream

operations can generate part of the answer given only partial information about the arguments. For `integral`, the first element of the output stream is the specified `initial_value`. Thus, we can generate the first element of the output stream without evaluating the integrand `dy`. Once we know the first element of `y`, the `stream_map` in the second line of `solve` can begin working to generate the first element of `dy`, which will produce the next element of `y`, and so on.

To take advantage of this idea, we will redefine `integral` to expect the integrand stream to be a *delayed argument*. The function `integral` will force the integrand to be evaluated only when it is required to generate more than the first element of the output stream:

```
function integral(delayed_integrand, intial_value, dt) {
    const integrand = delayed_integrand();
    const integ =
        pair(intial_value,
            add_streams(scale_stream(integrand, dt), int));
}
```

Now we can implement our `solve` function by delaying the evaluation of `dy` in the definition of `y`:

```
function solve(f, y0, dt) {
    const y = integral( () => dy, y0, dt);
    const dy = stream_map(f, y);
    return y;
}
```

In general, every caller of `integral` must now delay the integrand argument. We can demonstrate that the `solve` function works by approximating $e \approx 2.718$ by computing the value at $y = 1$ of the solution to the differential equation $dy/dt = y$ with initial condition $y(0) = 1$:

```
stream_ref(solve(y => y, 1, 0.001), 1000);
```

Exercise 3.77

The `integral` function used above was analogous to the 'implicit' definition of the infinite stream of integers in section 3.5.2. Alternatively, we can give a definition of `integral` that is more like `integers-starting-from` (also in section 3.5.2):

```
function integral(integrand, intial_value, dt) {
    return pair(intial_value,
        is_null(integrand) ? null
            : integral(stream_tail(integrand),
                dt * head(integrand) + initial_value,
                dt));
}
```

When used in systems with loops, this function has the same problem as does our original version of `integral`. Modify the function so that it expects the `integrand` as a delayed argument and hence can be used in the `solve` function shown above.

Exercise 3.78

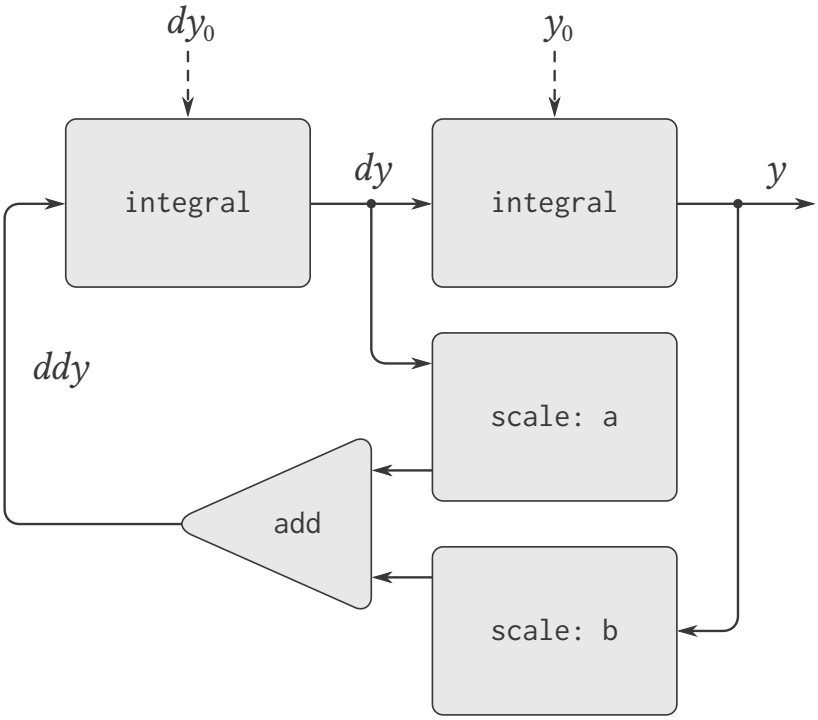


Figure 3.35: Signal-flow diagram for the solution to a second-order linear differential equation.

Consider the problem of designing a signal-processing system to study the homogeneous second-order linear differential equation

$$\frac{d^2y}{dt^2} - a\frac{dy}{dt} - by = 0$$

The output stream, modeling y, is generated by a network that contains a loop. This is because the value of d^2y/dt^2 depends upon the values of y and dy/dt and both of these are determined by integrating d^2y/dt^2. The diagram we would like to encode is shown in Figure 3.35. Write a function `solve_2nd` that takes as arguments the constants a, b, and dt and the initial values y_0 and dy_0 for y and dy/dt and generates the stream of successive values of y.

Exercise 3.79

Generalize the `solve-2nd` function of exercise 3.78 so that it can be used to solve general second-order differential equations $d^2y/dt^2 = f(dy/dt, y)$.

Exercise 3.80

A *series RLC circuit* consists of a resistor, a capacitor, and an inductor connected in series, as shown in Figure 3.36. If R, L, and C are the resistance, inductance, and capacitance, then the relations between voltage (v) and current (i) for the three components are described by the equations

$$\begin{aligned} v_R &= i_R R \\ v_L &= L\frac{di_L}{dt} \\ i_C &= C\frac{dv_C}{dt} \end{aligned}$$

and the circuit connections dictate the relations

$$\begin{aligned} i_R &= i_L = -i_C \\ v_C &= v_L + v_R \end{aligned}$$

Combining these equations shows that the state of the circuit (summarized by v_C, the voltage across the capacitor, and i_L, the current in the inductor) is described by the pair of differential equations

$$\begin{aligned} \frac{dv_C}{dt} &= -\frac{i_L}{C} \\ \frac{di_L}{dt} &= \frac{1}{L}v_C - \frac{R}{L}i_L \end{aligned}$$

The signal-flow diagram representing this system of differential equations is shown in Figure 3.37.

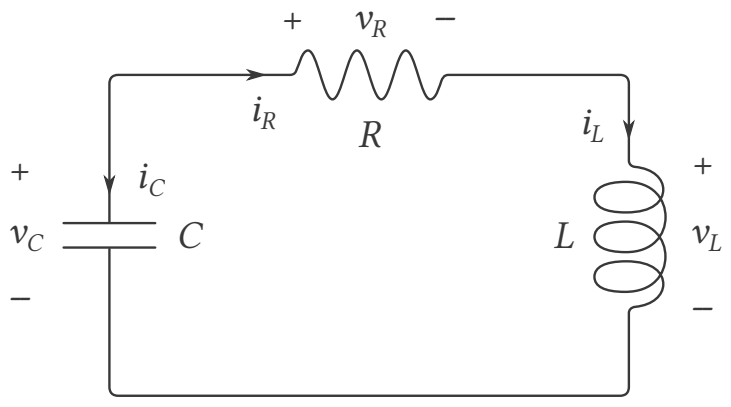


Figure 3.36: A series RLC circuit.

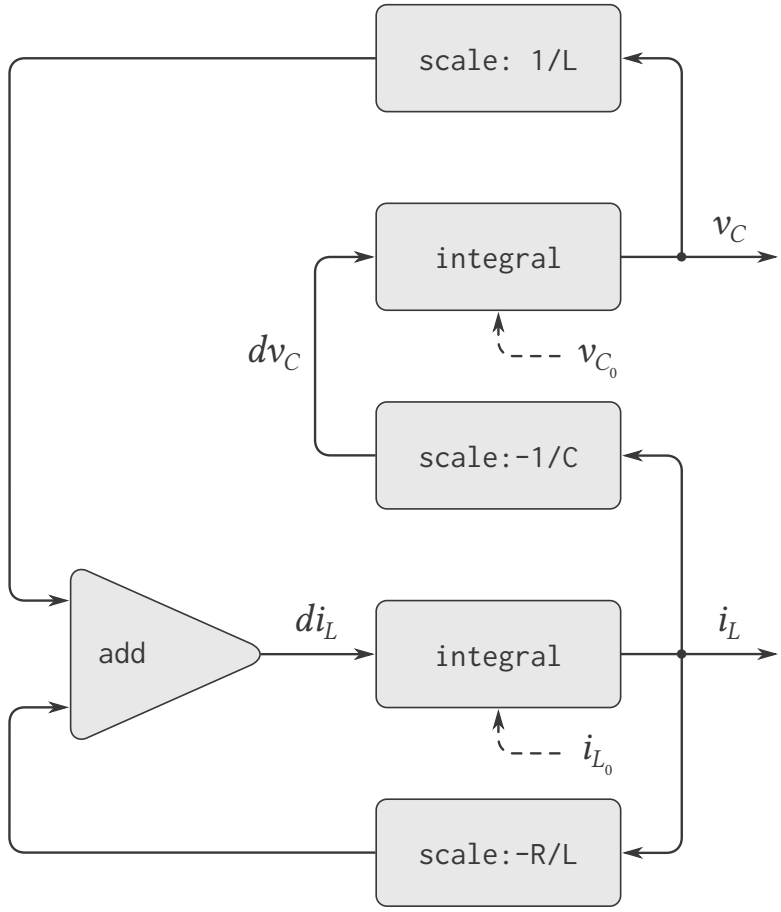


Figure 3.37: A signal-flow diagram for the solution to a series RLC circuit.

Write a function `RLC` that takes as arguments the parameters R, L, and C of the circuit and the time increment dt. In a manner similar to that of the `RC` function of exercise 3.73, `RLC` should produce a function that takes the initial values of the state variables, v_{C_0} and i_{L_0}, and produces a pair (using `pair`) of the streams of states v_C and i_L. Using `RLC`, generate the pair of streams that models the behavior of a series RLC circuit with $R = 1$ ohm, $C = 0.2$ farad, $L = 1$ henry, $dt = 0.1$ second, and initial values $i_{L_0} = 0$ amps and $v_{C_0} = 10$ volts.

Normal-order evaluation

The examples in this section illustrate how delayed evaluation provides great programming flexibility, but the same examples also show how this can make our programs more complex. Our new `integral` function, for instance, gives us the power to model systems with loops, but we must now remember that `integral` should be called with a delayed integrand, and every function that uses `integral` must be aware of this. In effect, we have created two classes of functions: ordinary functions and functions that take delayed arguments. In general, creating separate classes of functions forces us to create separate classes of higher-order functions as

well.[67]

One way to avoid the need for two different classes of functions is to make all functions take delayed arguments. We could adopt a model of evaluation in which all arguments to functions are automatically delayed and arguments are forced only when they are actually needed (for example, when they are required by a primitive operation). This would transform our language to use normal-order evaluation, which we first described when we introduced the substitution model for evaluation in section **??**. Converting to normal-order evaluation provides a uniform and elegant way to simplify the use of delayed evaluation, and this would be a natural strategy to adopt if we were concerned only with stream processing. In section 4.2, after we have studied the evaluator, we will see how to transform our language in just this way. Unfortunately, including delays in function calls wreaks havoc with our ability to design programs that depend on the order of events, such as programs that use assignment, mutate data, or perform input or output. Even a single delay in the tail of a pair can cause great confusion, as illustrated by exercise 3.51 and 3.52. As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, and devising ways to deal with both of these at once is an active area of research.

3.5.5 Modularity of Functional Programs and Modularity of Objects

Note: this section is a work in progress!

As we saw in section 3.1.2, one of the major benefits of introducing assignment is that we can increase the modularity of our systems by encapsulating, or 'hiding,' parts of the state of a large system within local variables. Stream models can provide an equivalent modularity without the use of assignment. As an illustration, we can reimplement the Monte Carlo estimation of π, which we examined in section 3.1.2, from a stream-processing point of view.

The key modularity issue was that we wished to hide the internal state of a random-number generator from programs that used random numbers. We began with a function `rand_update`, whose successive values furnished our supply of random numbers, and used this to produce a random-number generator:

[67] This is a small reflection, in JavaScript, of the difficulties that conventional strongly typed languages such as Pascal have in coping with higher-order functions. In such languages, the programmer must specify the data types of the arguments and the result of each function: number, logical value, sequence, and so on. Consequently, we could not express an abstraction such as 'map a given function `fun` over all the elements in a sequence' by a single higher-order function such as `stream_map`. Rather, we would need a different mapping function for each different combination of argument and result data types that might be specified for a `fun`. Maintaining a practical notion of 'data type' in the presence of higher-order functions raises many difficult issues. One way of dealing with this problem is illustrated by the language ML (Gordon, Milner, and Wadsworth 1979), whose 'polymorphic data types' include templates for higher-order transformations between data types. Moreover, data types for most functions in ML are never explicitly declared by the programmer. Instead, ML includes a *type-inferencing* mechanism that uses information in the environment to deduce the data types for newly defined functions.

```
function make_rand() {
    let x = random_init;
    function rand() {
        x = rand_update(x);
        return x;
    }
    return rand;
}
const rand = make_rand();
```

In the stream formulation there is no random-number generator *per se*, just a stream of random numbers produced by successive calls to `rand_update`:

We use this to construct the stream of outcomes of the Cesàro experiment performed on consecutive pairs in the `random_numbers` stream:

The `cesaro_stream` is now fed to a `monte_carlo` function, which produces a stream of estimates of probabilities. The results are then converted into a stream of estimates of π. This version of the program doesn't need a parameter telling how many trials to perform. Better estimates of π (from performing more experiments) are obtained by looking farther into the `pi` stream:

There is considerable modularity in this approach, because we still can formulate a general `monte_carlo` function that can deal with arbitrary experiments. Yet there is no assignment or local state.

Exercise 3.81

Exercise 3.6 discussed generalizing the random-number generator to allow one to reset the random-number sequence so as to produce repeatable sequences of 'random' numbers. Produce a stream formulation of this same generator that operates on an input stream of requests to `generate` a new random number or to `reset` the sequence to a specified value and that produces the desired stream of random numbers. Don't use assignment in your solution.

Exercise 3.82

Redo exercise 3.5 on Monte Carlo integration in terms of streams. The stream version of `estimate_integral` will not have an argument telling how many trials to perform. Instead, it will produce a stream of estimates based on successively more trials.

A functional-programming view of time

Let us now return to the issues of objects and state that were raised at the beginning of this chapter and examine them in a new light. We introduced assignment and mutable objects to provide a mechanism for modular construction of programs that model systems with state. We constructed computational objects with local state variables and used assignment to modify these variables. We modeled the temporal behavior of the objects in the world by the temporal behavior of the corresponding computational objects.

Now we have seen that streams provide an alternative way to model objects with local state. We can model a changing quantity, such as the local state of some object, using a stream that represents the time history of successive states. In essence, we represent time explicitly, using streams, so that we decouple time in our simulated world from the sequence of events that take place during evaluation. Indeed, because of the presence of `delay` there may be little relation between simulated time in the model and the order of events during the evaluation.

In order to contrast these two approaches to modeling, let us reconsider the implementation of a 'withdrawal processor' that monitors the balance in a bank account. In section 3.1.3 we implemented a simplified version of such a processor:

```
function make_simplified_withdraw(balance) {
    function withdraw(amount) {
        balance = balance - amount;
        return balance;
    }
    return withdraw;
}
```

Calls to `make_simplified_withdraw` produce computational objects, each with a local state variable `balance` that is decremented by successive calls to the object. The object takes an `amount` as an argument and returns the new balance. We can imagine the user of a bank account typing a sequence of inputs to such an object and observing the sequence of returned values shown on a display screen.

Alternatively, we can model a withdrawal processor as a function that takes as input a balance and a stream of amounts to withdraw and produces the stream of successive balances in the account:

```
function stream_withdraw(balance, amount_stream) {
    return pair(balance,
                () => stream_withdraw(
                          balance - head(amount_stream),
                          stream_tail(amount_stream)));
}
```

The function `stream_withdraw` implements a well-defined mathematical function whose out-

put is fully determined by its input. Suppose, however, that the input `amount_stream` is the stream of successive values typed by the user and that the resulting stream of balances is displayed. Then, from the perspective of the user who is typing values and watching results, the stream process has the same behavior as the object created by `make_simplified_withdraw`. However, with the stream version, there is no assignment, no local state variable, and consequently none of the theoretical difficulties that we encountered in section 3.1.3. Yet the system has state!

This is really remarkable. Even though `stream_withdraw` implements a well-defined mathematical function whose behavior does not change, the user's perception here is one of interacting with a system that has a changing state. One way to resolve this paradox is to realize that it is the user's temporal existence that imposes state on the system. If the user could step back from the interaction and think in terms of streams of balances rather than individual transactions, the system would appear stateless.[68]

From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write programs that model this kind of natural decomposition in our world (as we see it from our viewpoint as a part of that world) with structures in our computer, we make computational objects that are not functional—they must change with time. We model state with local state variables, and we model the changes of state with assignments to those variables. By doing this we make the time of execution of a computation model time in the world that we are part of, and thus we get 'objects' in our computer.

Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are part. However, as we've seen repeatedly throughout this chapter, these models raise thorny problems of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of *functional programming languages*, which do not include any provision for assignment or mutable data. In such a language, all functions implement well-defined mathematical functions of their arguments, whose behavior does not change. The functional approach is extremely attractive for dealing with concurrent systems.[69]

On the other hand, if we look closely, we can see time-related problems creeping into functional models as well. One particularly troublesome area arises when we wish to design interactive systems, especially ones that model interactions between independent entities. For instance, consider once more the implementation a banking system that permits joint bank accounts. In

[68] Similarly in physics, when we observe a moving particle, we say that the position (state) of the particle is changing. However, from the perspective of the particle's world line in space-time there is no change involved.

[69] John Backus, the inventor of Fortran, gave high visibility to functional programming when he was awarded the ACM Turing award in 1978. His acceptance speech (Backus 1978) strongly advocated the functional approach. A good overview of functional programming is given in Henderson 1980 and in Darlington, Henderson, and Turner 1982.

a conventional system using assignment and objects, we would model the fact that Peter and Paul share an account by having both Peter and Paul send their transaction requests to the same bank-account object, as we saw in section 3.1.3. From the stream point of view, where there are no 'objects' *per se*, we have already indicated that a bank account can be modeled as a process that operates on a stream of transaction requests to produce a stream of responses. Accordingly, we could model the fact that Peter and Paul have a joint bank account by merging Peter's stream of transaction requests with Paul's stream of requests and feeding the result to the bank-account stream process, as shown in figure 3.38.

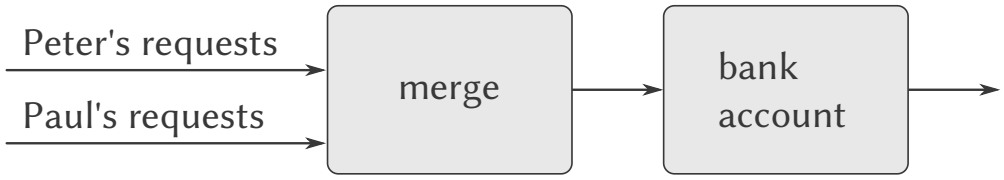


Figure 3.38: A joint bank account, modeled by merging two streams of transaction requests.

The trouble with this formulation is in the notion of *merge*. It will not do to merge the two streams by simply taking alternately one request from Peter and one request from Paul. Suppose Paul accesses the account only very rarely. We could hardly force Peter to wait for Paul to access the account before he could issue a second transaction. However such a merge is implemented, it must interleave the two transaction streams in some way that is constrained by 'real time' as perceived by Peter and Paul, in the sense that, if Peter and Paul meet, they can agree that certain transactions were processed before the meeting, and other transactions were processed after the meeting.[70]

This is precisely the same constraint that we had to deal with in section 3.4.1, where we found the need to introduce explicit synchronization to ensure a 'correct' order of events in concurrent processing of objects with state. Thus, in an attempt to support the functional style, the need to merge inputs from different agents reintroduces the same problems that the functional style was meant to eliminate.

We began this chapter with the goal of building computational models whose structure matches our perception of the real world we are trying to model. We can model the world as a collection of separate, time-bound, interacting objects with state, or we can model the world as a single, timeless, stateless unity. Each view has powerful advantages, but neither view alone is completely satisfactory. A grand unification has yet to emerge.[71]

[70] Observe that, for any two streams, there is in general more than one acceptable order of interleaving. Thus, technically, 'merge' is a relation rather than a function—the answer is not a deterministic function of the inputs. We already mentioned (footnote 38) that nondeterminism is essential when dealing with concurrency. The merge relation illustrates the same essential nondeterminism, from the functional perspective.

[71] The object model approximates the world by dividing it into separate pieces. The functional model does not modularize along object boundaries. The object model is useful when the unshared state of the 'objects' is much larger than the state that they share. An example of a place where the object viewpoint fails is quantum mechanics,

where thinking of things as individual particles leads to paradoxes and confusions. Unifying the object view with the functional view may have little to do with programming, but rather with fundamental epistemological issues.

Chapter 4

Metalinguistic Abstraction

> . . . It's in words that the magic is—Abracadabra, Open Sesame, and the rest—but the magic words in one story aren't magical in the next. The real magic is to understand which words work, and when, and for what; the trick is to learn the trick.
> . . . And those words are made from the letters of our alphabet: a couple-dozen squiggles we can draw with the pen. This is the key! And the treasure, too, if we can only get our hands on it! It's as if—as if the key to the treasure *is* the treasure!
>
> — John Barth, Chimera

In our study of program design, we have seen that expert programmers control the complexity of their designs with the same general techniques used by designers of all complex systems. They combine primitive elements to form compound objects, they abstract compound objects to form higher-level building blocks, and they preserve modularity by adopting appropriate large-scale views of system structure. In illustrating these techniques, we have used JavaScript as a language for describing processes and for constructing computational data objects and processes to model complex phenomena in the real world. However, as we confront increasingly complex problems, we will find that JavaScript, or indeed any fixed programming language, is not sufficient for our needs. We must constantly turn to new languages in order to express our ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in engineering design; we can often enhance our ability to deal with a complex problem by adopting a new language that enables us to describe (and hence to think about) the problem in a different way, using primitives, means of combination, and means of abstraction that are particularly well suited to the problem at hand.[1]

[1] The same idea is pervasive throughout all of engineering. For example, electrical engineers use many different languages for describing circuits. Two of these are the language of electrical *networks* and the language of

Programming is endowed with a multitude of languages. There are physical languages, such as the machine languages for particular computers. These languages are concerned with the representation of data and control in terms of individual bits of storage and primitive machine instructions. The machine-language programmer is concerned with using the given hardware to erect systems and utilities for the efficient implementation of resource-limited computations. High-level languages, erected on a machine-language substrate, hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. These languages have means of combination and abstraction, such as function definition, that are appropriate to the larger-scale organization of systems.

Metalinguistic abstraction—establishing new languages—plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An *evaluator* (or *interpreter*) for a programming language is a function that, when applied to an expression of the language, performs the actions required to evaluate that expression.

It is no exaggeration to regard this as the most fundamental idea in programming:

> The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others.

In fact, we can regard almost any program as the evaluator for some language. For instance, the polynomial manipulation system of section **??** embodies the rules of polynomial arithmetic and implements them in terms of operations on list-structured data. If we augment this system with functions to read and print polynomial expressions, we have the core of a special-purpose language for dealing with problems in symbolic mathematics. The digital-logic simulator of section 3.3.4 and the constraint propagator of section 3.3.5 are legitimate languages in their own right, each with its own primitives, means of combination, and means of abstraction. Seen from this perspective, the technology for coping with large-scale computer systems merges

electrical *systems*. The network language emphasizes the physical modeling of devices in terms of discrete electrical elements. The primitive objects of the network language are primitive electrical components such as resistors, capacitors, inductors, and transistors, which are characterized in terms of physical variables called voltage and current. When describing circuits in the network language, the engineer is concerned with the physical characteristics of a design. In contrast, the primitive objects of the system language are signal-processing modules such as filters and amplifiers. Only the functional behavior of the modules is relevant, and signals are manipulated without concern for their physical realization as voltages and currents. The system language is erected on the network language, in the sense that the elements of signal-processing systems are constructed from electrical networks. Here, however, the concerns are with the large-scale organization of electrical devices to solve a given application problem; the physical feasibility of the parts is assumed. This layered collection of languages is another example of the stratified design technique illustrated by the picture language of section 2.1.4.

with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages.

We now embark on a tour of the technology by which languages are established in terms of other languages. In this chapter we shall use JavaScript as a base, implementing evaluators as JavaScript functions. JavaScript is particularly well suited to this task, because of its ability to represent and manipulate symbolic expressions. We will take the first step in understanding how languages are implemented by building an evaluator for JavaScript itself. The language implemented by our evaluator will be a subset of JavaScript. Although the evaluator described in this chapter is written for a particular subset of JavaScript, it contains the essential structure of an evaluator for any expression-oriented language designed for writing programs for a sequential machine. (In fact, most language processors contain, deep within them, a little evaluator.) The evaluator has been simplified for the purposes of illustration and discussion, and some features have been left out that would be important to include in a production-quality JavaScript system. Nevertheless, this simple evaluator is adequate to execute most of the programs in this book.[2]

An important advantage of making the evaluator accessible as a JavaScript program is that we can implement alternative evaluation rules by describing these as modifications to the evaluator program. One place where we can use this power to good effect is to gain extra control over the ways in which computational models embody the notion of time, which was so central to the discussion in chapter 3. There, we mitigated some of the complexities of state and assignment by using streams to decouple the representation of time in the world from time in the computer. Our stream programs, however, were sometimes cumbersome, because they were constrained by the applicative-order evaluation of JavaScript. In section 4.2, we'll change the underlying language to provide for a more elegant approach, by modifying the evaluator to provide for *normal-order evaluation*.

4.1 The Metacircular Evaluator

Our evaluator for JavaScript will be implemented as a JavaScript program. It may seem circular to think about evaluating JavaScript programs using an evaluator that is itself implemented in JavaScript. However, evaluation is a process, so it is appropriate to describe the evaluation process using JavaScript, which, after all, is our tool for describing processes.[3] An evaluator

[2] The most important features that our evaluator leaves out are mechanisms for handling errors and supporting debugging. For a more extensive discussion of evaluators, see Friedman, Wand, and Haynes 1992, which gives an exposition of programming languages that proceeds via a sequence of evaluators written in the Scheme dialect of Lisp.

[3] Even so, there will remain important aspects of the evaluation process that are not elucidated by our evaluator. The most important of these are the detailed mechanisms by which functions call other functions and return values to their callers. We will address these issues in chapter 5, where we take a closer look at the evaluation

that is written in the same language that it evaluates is said to be *metacircular*.

The metacircular evaluator is essentially a JavaScript formulation of the environment model of evaluation described in section 3.2. Recall that the model has three basic parts:

- To evaluate an operator combination, evaluate the subexpressions and then apply the operator to the values of the subexpressions.
- To evaluate a function application combination, evaluate the function subexpression and the argument subexpressions, and then apply the value of the function subexpression to the values of the argument subexpressions.
- To apply a function to a set of arguments, evaluate the body of the function in a new environment. To construct this environment, extend the environment part of the function object by a frame in which the formal parameters of the function are bound to the arguments to which the function is applied.

These three rules describe the essence of the evaluation process, a basic cycle in which statements to be evaluated in environments are reduced to functions to be applied to arguments, which in turn are reduced to new statements to be evaluated in new environments, and so on, until we get down to symbols, whose values are looked up in the environment, and to operators, which are applied directly (see Figure 4.1).[4] This evaluation cycle will be embodied by the interplay between the two critical functions in the evaluator, `eval` and `apply`, which are described in section 4.1.1 (see Figure 4.1).

The implementation of the evaluator will depend upon functions that define the *syntax* of the expressions to be evaluated. We will use data abstraction to make the evaluator independent

process by implementing the evaluator as a simple register machine.

[4] If we grant ourselves the ability to apply primitives, then what remains for us to implement in the evaluator? The job of the evaluator is not to specify the primitives of the language, but rather to provide the connective tissue—the means of combination and the means of abstraction—that binds a collection of primitives to form a language. Specifically:

- The evaluator enables us to deal with nested expressions. For example, although simply applying primitives would suffice for evaluating the statement `1 + 6;`, it is not adequate for handling `1 + (2 * 3);`. As far as the primitive function `+` is concerned, its arguments must be numbers, and it would choke if we passed it the expression `2 * 3` as an argument. One important role of the evaluator is to choreograph function composition so that `2 * 3` is reduced to 6 before being passed as an argument to `+`.
- The evaluator allows us to use variables. For example, the primitive function for addition has no way to deal with expressions such as `x + 1`. We need an evaluator to keep track of variables and obtain their values before invoking the primitive functions.
- The evaluator allows us to define compound functions. This involves keeping track of function definitions, knowing how to use these definitions in evaluating expressions, and providing a mechanism that enables functions to accept arguments.
- The evaluator provides the other constructs of the language such as sequential composition and conditional expressions.

of the representation of the language. For example, rather than committing to a choice that an assignment is to be represented by a list beginning with the symbol assignment we use an abstract predicate `is_assignment` to test for an assignment, and we use abstract selectors `assignment_name` and `assignment_right_hand_side` to access the parts of an assignment. Implementation of expressions will be described in detail in section 4.1.2. There are also operations, described in section 4.1.3, that specify the representation of functions and environments. For example, `make_function_object` constructs compound functions, `lookup_name_value` accesses the values of variables, and `apply_builtin_function` applies a primitive function to a given list of arguments.

4.1.1 The Core of the Evaluator

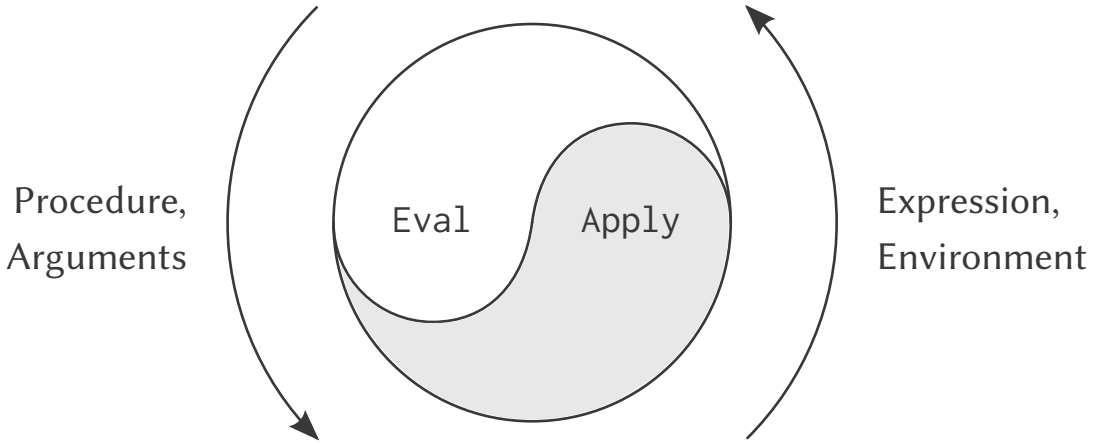


Figure 4.1: The `evaluate–apply` cycle exposes the essence of a computer language.

The evaluation process can be described as the interplay between two functions: `evaluate` and `apply`.

The function `evaluate`

The function `evaluate` takes as arguments a statement and an environment. It classifies the statement and directs its evaluation. The function `evaluate` is structured as a case analysis of the syntactic type of the expression to be evaluated. In order to keep the function general, we express the determination of the type of a statement abstractly, making no commitment to any particular representation for the various types of statements. Each type of statement has a predicate that tests for it and an abstract means for selecting its parts. This *abstract syntax* makes it easy to see how we can change the syntax of the language by using the same evaluator, but with a different collection of syntax functions.

Here is the definition of `evaluate`:

```
function evaluate(stmt, env) {
   return is_self_evaluating(stmt)
          ?  stmt
        : is_name(stmt)
          ? lookup_name_value(name_of_name(stmt), env)
        : is_constant_declaration(stmt)
          ? eval_constant_declaration(stmt, env)
        : is_variable_declaration(stmt)
          ? eval_variable_declaration(stmt, env)
        : is_assignment(stmt)
          ? eval_assignment(stmt, env)
        : is_conditional_expression(stmt)
          ? eval_conditional_expression(stmt, env)
        : is_function_definition(stmt)
          ? eval_function_definition(stmt, env)
        : is_sequence(stmt)
          ? eval_sequence(sequence_statements(stmt), env)
        : is_block(stmt)
          ? eval_block(stmt, env)
        : is_return_statement(stmt)
          ? eval_return_statement(stmt, env)
        : is_application(stmt)
          ? apply(evaluate(operator(stmt), env),
                  list_of_values(operands(stmt), env))
        : error(stmt, "Unknown statement type in evaluate: ");
}
```

For clarity, `evaluate` has been implemented as a case analysis using conditional expressions. The disadvantage of this is that our function handles only a few distinguishable types of statements, and no new ones can be defined without editing the definition of `evaluate`. In most interpreter implementations, dispatching on the type of an expression is done in a data-directed style. This allows a user to add new types of statements and expressions that `evaluate` can distinguish, without modifying the definition of `evaluate` itself. (See exercise 4.2.)

Apply

The function `apply` takes two arguments, a function and a list of arguments to which the function should be applied. The function `apply` classifies functions into two kinds: It calls `apply_primitive_function` to apply primitives; it applies compound functions by sequentially evaluating the statements that make up the body of the function. The environment for the evaluation of the body of a compound function is constructed by extending the base environment carried by the function to include a frame that binds the parameters of the function to the arguments to which the function is to be applied, and the body's local names to a special value `no_value_yet`. The function `local_names` for computing the body's local names is also

used for *blocks* and explained below. Here is the definition of `apply`:

```
function apply(fun, args) {
   if (is_primitive_function(fun)) {
      return apply_primitive_function(fun, args);
   } else if (is_compound_function(fun)) {
      const body = function_body(fun);
      const locals = local_names(body);
      const names = insert_all(function_parameters(fun),
                               locals);
      const temp_values = map(x => no_value_yet,
                              locals);
      const values = append(args,
                            temp_values);
      const result =
         evaluate(body,
                  extend_environment(
                      names,
                      values,
                      function_environment(fun)));
      if (is_return_value(result)) {
         return return_value_content(result);
      } else {
          return undefined;
      }
   } else {
      error(fun, "Unknown function type in apply");
   }
}
```

In order to return a value, JavaScript functions need to evaluate a return statement. If a function terminates without return, the value `undefined` is returned. Thus, if the evaluation of the function body yields a return value, the content of the return value is retrieved, and otherwise the value `undefined` is returned.

Function arguments

When `evaluate` processes a function application, it uses `list_of_values` to produce the list of arguments to which the function is to be applied. The function `list_of_values` takes as an argument the operands of the combination. It evaluates each operand and returns a list of the corresponding values:[5]

[5] We could have simplified the `is_application` clause in `evaluate` by using `map` (and stipulating that `operands` returns a list) rather than writing an explicit `list_of_values` function. We chose not to use `map` here to emphasize the fact that the evaluator can be implemented without any use of higher-order functions (and thus could be written in a language that doesn't have higher-order functions), even though the language that it supports will include higher-order functions.

```
function list_of_values(exps, env) {
    if (no_operands(exps)) {
        return null;
    } else {
        return pair(evaluate(first_operand(exps), env),
                    list_of_values(rest_operands(exps), env));
   }
}
```

Conditionals

The function `eval_conditional_expression` evaluates the predicate part of an conditional expression in the given environment. If the result is true, the consequent is evaluated, otherwise the alternative:

```
function eval_conditional_expression(stmt, env) {
    return is_true(evaluate(cond_expr_pred(stmt),
                            env))
           ? evaluate(cond_expr_cons(stmt),
                      env)
           : evaluate(cond_expr_alt(stmt),
                      env);
}
```

The use of `is_true` in `eval_conditional_expression` highlights the issue of the connection between an implemented language and an implementation language. The predicate is evaluated in the language being implemented and thus yields a value in that language. The interpreter predicate `is_true` translates that value into a value that can be tested by the conditional expression in the implementation language: The metacircular representation of truth might not be the same as that of the underlying JavaScript.[6]

Function definitions

The evaluation of a function definition expression creates a function object that represents the function during the evaluation. The function object contains the parameters and the body of the function definition, as well as the environment with respect to which the function definition is evaluated. According to the environment model, this is the environment that needs to be extended, when the function gets applied to actual arguments.

[6] In this case, the language being implemented and the implementation language are the same. Contemplation of the meaning of `is_true` here yields expansion of consciousness without the abuse of substance.

```
function eval_function_definition(stmt,env) {
    return make_compound_function(
              map(name_of_name,
                  function_definition_parameters(stmt)),
              function_definition_body(stmt),
              env);
}
```

Sequences

The function `eval_sequence` is used by `eval` to evaluate a sequence of statements. Note that the evaluation of the first component of a sequence may yield a return value, in which case the rest of the statement is not evaluated.

```
function eval_sequence(stmts, env) {
    if (is_empty_sequence(stmts)) {
        return undefined;
    } else if (is_last_statement(stmts)) {
            return evaluate(first_statement(stmts),env);
    } else {
        const first_stmt_value =
            evaluate(first_statement(stmts),env);
        if (is_return_value(first_stmt_value)) {
            return first_stmt_value;
        } else {
            return eval_sequence(
                rest_statements(stmts),env);
        }
    }
}
```

Blocks

The function `eval_block` is used by `evaluate` to evaluate block statements. The constants and variables declared in the block need to be local to the block. The evaluation of block statements evaluates the body of the block with respect to an environment that extends the current environment with a binding of the local names of the block body to a special value `no_value_yet`.

```
function eval_block(stmt, env) {
    const body = block_body(stmt);
    const locals = local_names(body);
    const temp_values = map(x => no_value_yet,
                            locals);
    return evaluate(body,
```

```
                extend_environment(locals, temp_values, env));
}
```

The function `local_names` collects all names declared in the body statements. For a name to be included in the list of `local_names`, it needs to be declared outside of any other block or function.

```
function local_names(stmt) {
    if (is_sequence(stmt)) {
        const stmts = sequence_statements(stmt);
        return is_empty_sequence(stmts)
            ? null
            : insert_all(
                  local_names(first_statement(stmts)),
                  local_names(make_sequence(
                              rest_statements(stmts))));
    } else {
       return is_constant_declaration(stmt)
           ? list(constant_declaration_name(stmt))
           : is_variable_declaration(stmt)
             ? list(variable_declaration_name(stmt))
             : null;
    }
}
```

```
local_names(parse("const x = 1; let y = 2;"));
```

Return statements

The function `eval_return_statement` is used by `evaluate` to evaluate return statements. As seen in the evaluation of sequences, the result of evaluation of return statements needs to be identifiable so that the evaluation of function bodies can return immediately, even if there are statements after the return statement. For this purpose, the evaluation of a return statement wraps the result of evaluating the return expression in a return value object.

```
function eval_return_statement(stmt, env) {
    return make_return_value(
               evaluate(return_statement_expression(stmt),
                        env));
}
```

Assignments and declarations

The following function handles assignments to variables. It calls `evaluate` to find the value to be assigned and transmits the variable and the resulting value to `assign_name_to_value` to be installed in the designated environment.

```
function eval_assignment(stmt, env) {
    const value = evaluate(assignment_value(stmt), env);
    assign_name_value(assignment_name(stmt), value, env);
    return value;
}
```

Declarations of constants and variables are handled in a similar manner. Section 4.1.3 explains how we distinguish variables and constants in the functions and how we prevent assignment to constants.

```
function eval_variable_declaration(stmt, env) {
    set_name_value(variable_declaration_name(stmt),
        evaluate(variable_declaration_value(stmt), env),
        env);
}
function eval_constant_declaration(stmt, env) {
    set_name_value(constant_declaration_name(stmt),
        evaluate(constant_declaration_value(stmt), env),
        env);
}
```

Note that the returned value of constant and variable declaration is the value `undefined`, as prescribed by the ECMAScript standard (Ecma 1997).

Exercise 4.1

Notice that we cannot tell whether the metacircular evaluator evaluates operands from left to right or from right to left. Its evaluation order is inherited from the underlying JavaScript: If the arguments to `pair` in `list_of_values` are evaluated from left to right, then `list_of_values` will evaluate operands from left to right; and if the arguments to `pair` are evaluated from right to left, then `list_of_values` will evaluate operands from right to left. Write a version of `list_of_values` that evaluates operands from left to right regardless of the order of evaluation in the underlying JavaScript. Also write a version of `list_of_values` that evaluates operands from right to left.

4.1.2 Representing Statements and Expressions

The evaluator is reminiscent of the symbolic differentiation program discussed in section **??**. Both programs operate on symbolic expressions. In both programs, the result of operating on a compound expression is determined by operating recursively on the pieces of the expression and combining the results in a way that depends on the type of the expression. In both programs we used data abstraction to decouple the general rules of operation from the details of how expressions are represented. In the differentiation program this meant that the same differentiation function could deal with algebraic expressions in prefix form, in infix form, or in some other form. For the evaluator, this means that the syntax of the language being evaluated is determined solely by the functions that classify and extract pieces of expressions.

Here is the specification of the syntax of our language:

- The self-evaluating items are numbers, strings and boolean values.

```
function is_self_evaluating(stmt) {
    return is_number(stmt) ||
           is_string(stmt) ||
           is_boolean(stmt);
}
```

- The function `is_name` tests whether the given statement is a name expression, and the function `name_of_name` accesses the JavaScript string that represents the name.

```
function is_name(stmt) {
    return is_tagged_list(stmt, "name");
}
function name_of_name(stmt) {
    return head(tail(stmt));
}
```

  The function `is_name` is defined in terms of the function `is_tagged_list`, which identifies lists beginning with a designated string that we call *tag*:

```
function is_tagged_list(stmt, the_tag) {
    return is_pair(stmt) && head(stmt) === the_tag;
}
```

- Assignments have the form *name* = *value*:

```
function is_assignment(stmt) {
   return is_tagged_list(stmt, "assignment");
}
function assignment_name(stmt) {
   return head(tail(head(tail(stmt))));
}
function assignment_value(stmt) {
```

```
    return head(tail(tail(stmt)));
}
```

– Declarations have the form

```
const name = value;
```

or

```
let name = value;
```

or

```
function name(parameter_1, ..., parameter_n) {
    body
}
```

Here, we treat the latter form (function declarations) as syntactic sugar[7] for

```
const name = (parameter_1, ..., parameter_n) => { body; };
```

The corresponding syntax functions are the following:

```
function is_constant_declaration(stmt) {
    return is_tagged_list(stmt, "constant_declaration");
}
function constant_declaration_name(stmt) {
    return head(tail(head(tail(stmt))));
}
function constant_declaration_value(stmt) {
    return head(tail(tail(stmt)));
}
function is_variable_declaration(stmt) {
    return is_tagged_list(stmt, "variable_declaration");
}
function variable_declaration_name(stmt) {
    return head(tail(head(tail(stmt))));
}
function variable_declaration_value(stmt) {
    return head(tail(tail(stmt)));
}
```

– Function definitions are objects tagged with the string `function_definition`:

```
function is_function_definition(stmt) {
    return is_tagged_list(stmt, "function_definition");
}
function function_definition_parameters(stmt) {
    return head(tail(stmt));
}
```

[7] In actual JavaScript, there is a subtle difference between the two forms. The interpretation of function declaration statements involves reordering of sequence statements, a topic which we prefer to skip at this point.

```
function function_definition_body(stmt) {
   return head(tail(tail(stmt)));
}
```

- **return** statements are objects tagged with the string "return_statement":

```
function is_return_statement(stmt) {
   return is_tagged_list(stmt, "return_statement");
}
function return_statement_expression(stmt) {
   return head(tail(stmt));
}
```

- Conditional expressions are tagged with "conditional_expression" and have a predicate, a consequent, and an alternative.

```
function is_conditional_expression(stmt) {
   return is_tagged_list(stmt,
                "conditional_expression");
}
function cond_expr_pred(stmt) {
   return list_ref(stmt, 1);
}
function cond_expr_cons(stmt) {
   return list_ref(stmt, 2);
}
function cond_expr_alt(stmt) {
   return list_ref(stmt, 3);
}
```

- A sequence is a list of statements.

```
function is_sequence(stmt) {
   return is_tagged_list(stmt, "sequence");
}
function make_sequence(stmts) {
   return list("sequence", stmts);
}
function sequence_statements(stmt) {
   return head(tail(stmt));
}
function is_empty_sequence(stmts) {
   return is_null(stmts);
}
function is_last_statement(stmts) {
   return is_null(tail(stmts));
}
function first_statement(stmts) {
   return head(stmts);
```

```
}
function rest_statements(stmts) {
   return tail(stmts);
}
```

– A block contains its body statement.

```
    function is_block(stmt) {
        return is_tagged_list(stmt, "block");
    }
    function make_block(stmt) {
       return list("block", stmt);
    }
function block_body(stmt) {
    return head(tail(stmt));
}
```

– A function application is an object tagged with the string `"application"`. We provide access functions for the `operator`, the `operands`, and three functions for iterating through the operand list:

```
    function is_application(stmt) {
       return is_tagged_list(stmt, "application");
    }
    function operator(stmt) {
       return head(tail(stmt));
    }
function operands(stmt) {
   return head(tail(tail(stmt)));
}
function no_operands(ops) {
   return is_null(ops);
}
function first_operand(ops) {
   return head(ops);
}
function rest_operands(ops) {
   return tail(ops);
}
```

Exercise 4.2

Rewrite `evaluate` so that the dispatch is done in data-directed style. Compare this with the data-directed differentiation function of exercise 2.37. (You may use the `head` of a compound expression as the type of the expression, as is appropriate for the syntax implemented in this section.)

Exercise 4.3

Recall the definitions of the special forms `&&` and `||` from chapter 1:

- *expression*$_1$ `&&` *expression*$_2$: The expression *expression*$_1$ is evaluated first. If it evaluates to `false`, `false` is returned; the expression *expression*$_2$ is not evaluated. If it evaluates to `true`, the value of *expression*$_2$ is returned.
- *expression*$_1$ `||` *expression*$_2$: The expression *expression*$_1$ is evaluated first. If it evaluates to `true`, `true` is returned; the expression *expression*$_2$ is not evaluated. If it evaluates to `false`, the value of *expression*$_2$ is returned.

Include `&&` and `||` expressions by defining appropriate syntax functions and evaluation functions `eval_and` and `eval_or`

4.1.3 Evaluator Data Structures

In addition to defining the external syntax of expressions, the evaluator implementation must also define the data structures that the evaluator manipulates internally, as part of the execution of a program, such as the representation of functions and environments and the representation of true and false.

Testing of predicates

To enter the consequent of a conditional, we expect the predicate to evaluate to the value `true`, and thus we define the evaluator function `is_true` as follows:

```
function is_true(x) {
    return x === true;
}
```

With the definition of the function `eval_conditional_expression` of section 4.1.1, this means that our evaluator evaluates the alternative statement for any predicate value other than `true`.

Representing functions

To handle primitives, we assume that we have available the following functions:

- `apply_primitive_function(` *fun*, *args* `)` applies the given primitive function to the argument values in the list *args* and returns the result of the application.
- `is_primitive_function(`*fun*`)` tests whether *fun* is a primitive function.

These mechanisms for handling primitives are further described in section 4.1.4.

Compound functions are constructed from parameters, function bodies, and environments using the constructor `make_compound_function`:

```
function make_compound_function(parameters, body, env) {
    return list("compound_function",
                parameters, body, env);
}
function is_compound_function(f) {
    return is_tagged_list(f, "compound_function");
}
function function_parameters(f) {
    return list_ref(f, 1);
}
function function_body(f) {
    return list_ref(f, 2);
}
function function_environment(f) {
    return list_ref(f, 3);
}
```

Representing return values

We saw in section 4.1.1 that the evaluation of sequences terminates with the first return statement encountered, and that the evaluation of function applications needs to return the value `undefined` if the evaluation of the function body does not encounter a **`return`** statement. In order to identify the evaluation of **`return`** statements, we introduce **`return`** values as evaluator data structures.

```
function make_return_value(content) {
    return list("return_value", content);
}
function is_return_value(value) {
    return is_tagged_list(value,"return_value");
}
function return_value_content(value) {
```

```
    return head(tail(value));
}
```

Operations on Environments

The evaluator needs operations for manipulating environments. As explained in section 3.2, an environment is a sequence of frames, where each frame is a table of bindings that associate names with their corresponding values. We use the following operations for manipulating environments:

- `lookup_name_value(` *name*, *env* `)` returns the value that is bound to the symbol *name* in the environment env, or signals an error if the name is unbound.
- `extend_environment(` *names*, *values*, *base-env* `)` returns a new environment, consisting of a new frame in which the symbols in the list *names* are bound to the corresponding elements in the list *values* (each tagged as *mutable*), where the enclosing environment is the environment *base-env*.
- `set_name_value(` *name*, *value*, *env* `)` sets the given *name* to the given *value* in the first frame of the environment *env*.
- `assign_name_value(` *name*, *value*, *env* `)` checks if the value associated to the name *name* is tagged as mutable, and if yes, changes its binding in the environment *env* so that the name is now bound to the value *value*, or signals an error if the name is unbound or its value tagged as immutable.

To implement these operations we represent an environment as a list of frames. The enclosing environment of an environment is the `tail` of the list. The empty environment is simply the empty list.

```
function enclosing_environment(env) {
    return tail(env);
}
function first_frame(env) {
    return head(env);
}
function enclose_by(frame, env) {
    return pair(frame, env);
}
const the_empty_environment = null;
function is_empty_environment(env) {
    return is_null(env);
}
```

Each frame of an environment is represented as a pair of lists: a list of the variables bound in that frame and a list of the associated values.[8]

```
function make_frame(names, values) {
    return pair(names, values);
}
function frame_names(frame) {
    return head(frame);
}
function frame_values(frame) {
    return tail(frame);
}
```

To extend an environment by a new frame that associates names with values, we make a frame consisting of the list of names and the list of values, and we adjoin this to the environment. We signal an error if the number of names does not match the number of values.

```
function extend_environment(names, vals, base_env) {
    if (length(names) === length(vals)) {
        return enclose_by(
                   make_frame(names,
                      map(x => pair(x, true), vals)),
                   base_env);
    } else if (length(names) < length(vals)) {
        error("Too many arguments supplied: " +
              stringify(names) + ", " +
              stringify(vals));
    } else {
        error("Too few arguments supplied: " +
              stringify(names) + ", " +
              stringify(vals));
    }
}
```

The function `extend_environment` is used by `apply` in section 4.1.1 to bind the parameters of a function to its arguments. In order to allow for assignment to function parameters, as in function `make_withdraw_with_balance` of section 3.1.1, we choose to tag the values in `extend_environment` as mutable, using `map`.

To look up a name in an environment, we scan the list of name in the first frame. If we find the desired name, we return the corresponding element in the list of values. If we do not find the name in the current frame, we search the enclosing environment, and so on. If we reach the empty environment, we signal an 'unbound name' error.

[8] Frames are not really a data abstraction in the following code: `assign_name_value` and `define_variable` use `set_head` to directly modify the values in a frame. The purpose of the frame functions is to make the environment-manipulation functions easy to read.

```
// We use a nullary function as temporary value for names whose
// declaration has not yet been evaluated. The purpose of the
// function definition is purely to create a unique identity;
// the function will never be applied and its return value
// (null) is irrelevant.
const no_value_yet = () => null;
```

```
function lookup_name_value(name, env) {
    function env_loop(env) {
        function scan(names, vals) {
            return is_null(names)
                   ? env_loop(
                       enclosing_environment(env))
                   : name === head(names)
                     ? head(head(vals))
                     : scan(tail(names), tail(vals));
        }
        if (is_empty_environment(env)) {
            error(name, "Unbound name: ");
        } else {
            const frame = first_frame(env);
            const value =  scan(frame_names(frame),
                                frame_values(frame));
            return value === no_value_yet
              ? error(name, "Name use before declaration: ")
              : value;
        }
    }
    return env_loop(env);
}
```

To assign a name to a new value in a specified environment, we scan for the name, just as in `lookup_name_value`, and change the corresponding value when we find it, provided it is tagged as mutable.

```
function assign_name_value(name, val, env) {
    function env_loop(env) {
        function scan(names, vals) {
            return is_null(names)
                ? env_loop(
                    enclosing_environment(env))
                : name === head(names)
                  ? ( tail(head(vals))
                      ? set_head(head(vals), val)
                      : error("no assignment " +
                          "to constants allowed") )
                  : scan(tail(names), tail(vals));
        }
```

```
        if (env === the_empty_environment) {
            error(name, "Unbound name in assignment: ");
        } else {
            const frame = first_frame(env);
            return scan(frame_names(frame),
                        frame_values(frame));
        }
    }
    return env_loop(env);
}
```

Function application and the evaluation of blocks already ensure that local names are included in the current environment before the actual declaration is evaluated. Therefore, the declaration of constants and variables can be handled similar to variable assignment, except that we can insist that the name is found in the innermost frame.

```
function set_name_value(name, val, env) {
    function scan(names, vals) {
        return is_null(names)
            ? error("internal error: name not found")
            : name === head(names)
              ? set_head(head(vals), val)
              : scan(tail(names), tail(vals));
    }
    const frame = first_frame(env);
    return scan(frame_names(frame),
                frame_values(frame));
}
```

The method described here is only one of many plausible ways to represent environments. Since we used data abstraction to isolate the rest of the evaluator from the detailed choice of representation, we could change the environment representation if we wanted to. (See exercise 4.4.) In a production-quality JavaScript system, the speed of the evaluator's environment operations—especially that of variable lookup—has a major impact on the performance of the system. The representation described here, although conceptually simple, is not efficient and would not ordinarily be used in a production system.[9]

Exercise 4.4

Instead of representing a frame as a pair of lists, we can represent a frame as a list of bindings, where each binding is a name-value pair. Rewrite the environment operations to use this alternative representation.

[9]The drawback of this representation (as well as the variant in exercise 4.4) is that the evaluator may have to search through many frames in order to find the binding for a given variable. (Such an approach is referred to as *deep binding*.) One way to avoid this inefficiency is to make use of a strategy called *lexical addressing*

Exercise 4.5

JavaScript allows us to create new bindings for names by means of constant and variable declaration, but provides no way to get rid of bindings. Implement for the evaluator a 'function' `make_unbound` that removes the binding of a name given as 'argument' from the environment in which the application of the function is evaluated. This problem is not completely specified. For example, should we remove only the binding in the first frame of the environment? Complete the specification and justify any choices you make.

4.1.4 Running the Evaluator as a Program

Given the evaluator, we have in our hands a description (expressed in JavaScript) of the process by which JavaScript expressions are evaluated. One advantage of expressing the evaluator as a program is that we can run the program. This gives us, running within JavaScript, a working model of how JavaScript itself evaluates expressions. This can serve as a framework for experimenting with evaluation rules, as we shall do later in this chapter.

Our evaluator program reduces expressions ultimately to the application of primitive functions. Therefore, all that we need to run the evaluator is to create a mechanism that calls on the underlying JavaScript system to model the application of primitive functions.

There must be a binding for each primitive function name, so that when `evaluate` evaluates the operator of an application of a primitive, it will find an object to pass to `apply`. We thus set up a global environment that associates unique objects with the names of the primitive functions that can appear in the expressions we will be evaluating. The global environment also includes bindings for the symbols `undefined`, `NaN` and `Infinity`, so that they can be used as constants in expressions to be evaluated.

```
function setup_environment() {
    const primitive_function_names =
        map(f => head(f), primitive_functions);
    const primitive_function_values =
        map(f => make_primitive_function(head(tail(f))),
            primitive_functions);
    const primitive_constant_names =
        map(f => head(f), primitive_constants);
    const primitive_constant_values =
        map(f => head(tail(f)),
            primitive_constants);
    return extend_environment(
               append(primitive_function_names,
                      primitive_constant_names),
               append(primitive_function_values,
                      primitive_constant_values),
```

```
                the_empty_environment);
}
```

It does not matter how we represent primitive functions, so long as `apply` can identify and apply them using the functions `is_primitive_function` and `apply_primitive_function`. We have chosen to represent a primitive function as a list beginning with the string `"primitive"` and containing a function in the underlying JavaScript that implements that primitive.

```
function make_primitive_function(impl) {
    return list("primitive", impl);
}
function is_primitive_function(fun) {
   return is_tagged_list(fun, "primitive");
}
function primitive_implementation(fun) {
   return list_ref(fun, 1);
}
```

The function `setup_environment` will get the primitive names and implementation functions from a list:[10]

```
const primitive_functions = list(
       list("display",       display          ),
       list("error",         error            ),
       list("+",             (x, y) => x + y  ),
       list("-",             (x, y) => x - y  ),
       list("*",             (x, y) => x * y  ),
       list("/",             (x, y) => x / y  ),
       list("%",             (x, y) => x % y  ),
       list("===",           (x, y) => x === y),
       list("!==",           (x, y) => x !== y),
       list("<",             (x, y) => x <   y),
       list("<=",            (x, y) => x <=  y),
       list(">",             (x, y) => x >   y),
       list(">=",            (x, y) => x >=  y),
       list("!",              x     =>   !   x)
       );
```

Similar to primitive functions, we define primitive values that are installed in the global environment by the function `setup_environment`.

[10] Any function defined in the underlying JavaScript can be used as a primitive for the metacircular evaluator. The name of a primitive installed in the evaluator need not be the same as the name of its implementation in the underlying JavaScript; the names are the same here because the metacircular evaluator implements JavaScript itself. Thus, for example, we could put `list("first", head)` or `list("square", x => x * x)` in the list of `primitive_functions`.

```
const primitive_constants = list(list("undefined", undefined),
                                 list("NaN",       NaN),
                                 list("Infinity",  Infinity),
                                 list("math_PI",   math_PI)
                                );
```

To apply a primitive function, we simply apply the implementation function to the arguments, using the underlying JavaScript system: [11]

```
function apply_primitive_function(fun, argument_list) {
    return apply_in_underlying_javascript(
                primitive_implementation(fun),
                argument_list);
}
```

For convenience in running the metacircular evaluator, we provide a *driver loop* that models the underlying JavaScript system. It prints a *prompt*, reads an input expression from a pop-up window, evaluates this expression in a suitable environment, and prints the result on the next pop-up window. We precede each printed result by an *output prompt* so as to distinguish the value of the expression from the output that may be printed.

```
const input_prompt = "M-Eval input: ";
const output_prompt = "M-Eval output: ";

function driver_loop(env, history) {
    const input = prompt(history + input_prompt);
    if (input === null) {
        display("session has ended");
    } else {
        const program = parse(input);
        const locals = local_names(program);
        const temp_values = map(x => no_value_yet,
                                locals);
        const new_env = extend_environment(locals, temp_values, env);
        const res = evaluate(program, new_env);
```

[11] JavaScript's apply method of function objects expects arguments in an array. Thus, the `argument_list` is transformed into an array using a **`while`** loop:

```
function apply_in_underlying_javascript(prim, argument_list) {
   const argument_array = [];
   let i = 0;
   while (!is_null(argument_list)) {
      argument_array[i] = head(argument_list);
      i = i + 1;
      argument_list = tail(argument_list);
   }
   return prim.apply(prim, argument_array);
}
```

We have made use of `apply_in_underlying_javascript` to define the function `apply` in section 2.2.3.

```
        driver_loop(new_env, history + "\\n" +
                             input_prompt + input + "\\n" +
                             output_prompt + stringify(user_print(res)));
    }
}
```

We transform the input string that is read by `prompt` into a tagged-list representation of the statement according to the description in section 4.1.2, a process called parsing and accomplished by the primitive function `parse`. Similar to the evaluation of blocks in 4.1.1, we create a new environment by extending the given environment by a binding of the local names in the program to the special value `no_value_yet`, and evaluate the program with respect to the new environment. Passing the new environment in the recursive call of `driver_loop` ensures that subsequent programs can refer to previously declared names.

We use a special printing function `user_print`, to avoid printing the environment part of a compound function, which may be a very long list (or may even contain cycles).

```
function user_print(object) {
   return is_compound_function(object)
       ? "compound function" +
         stringify(function_parameters(object)) +
         stringify(function_body(object)) +
         "<environment>"
       : object;
}
```

Now all we need to do to run the evaluator is to initialize the global environment and start the driver loop. Here is a sample interaction:

```
const the_global_environment = setup_environment();
driver_loop(the_global_environment, "");
```

```
M-Eval input:
function append(xs,ys) {
    if (is_null(xs)) {
       return ys;
    } else {
       return pair(head(xs),append(tail(xs),ys));
    }
}
M-Eval value: undefined

M-Eval input:
append(list('a', 'b', 'c'), list('d', 'e', 'f'));

M-Eval value: ['a', ['b', ['c', ['d', ['e', ['f', null]]]]]]
```

Exercise 4.6

Eva Lu Ator and Louis Reasoner are each experimenting with the metacircular evaluator. Eva types in the definition of `map`, and runs some test programs that use it. They work fine. Louis, in contrast, has installed the system version of `map` as a primitive for the metacircular evaluator. When he tries it, things go terribly wrong. Explain why Louis's `map` fails even though Eva's works.

4.1.5 Data as Programs

In thinking about a JavaScript program that evaluates JavaScript expressions, an analogy might be helpful. One operational view of the meaning of a program is that a program is a description of an abstract (perhaps infinitely large) machine. For example, consider the familiar program to compute factorials:

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

We may regard this program as the description of a machine containing parts that decrement, multiply, and test for equality, together with a two-position switch and another factorial machine. (The factorial machine is infinite because it contains another factorial machine within it.) Figure 4.2 is a flow diagram for the factorial machine, showing how the parts are wired together.

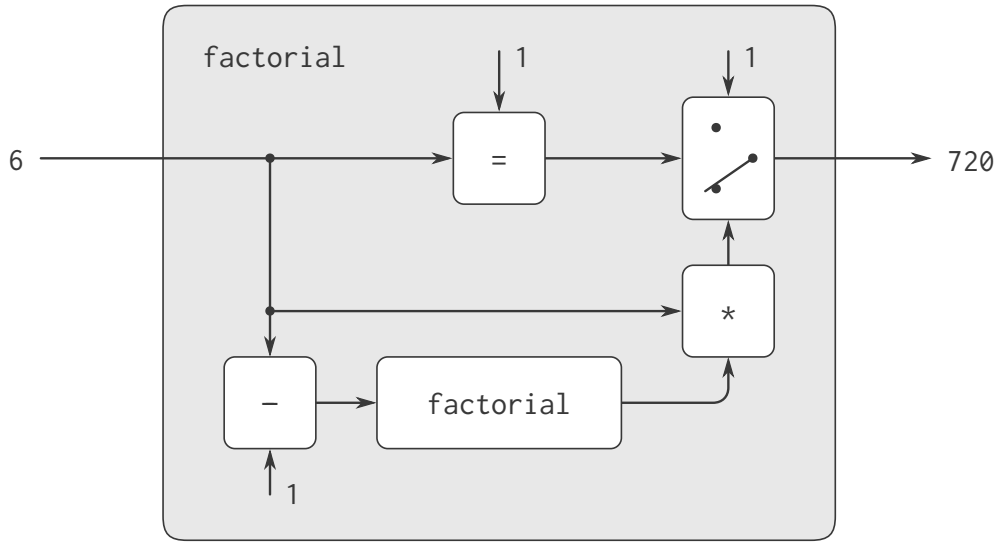


Figure 4.2: The factorial program, viewed as an abstract machine.

In a similar way, we can regard the evaluator as a very special machine that takes as input

a description of a machine. Given this input, the evaluator configures itself to emulate the machine described. For example, if we feed our evaluator the definition of `factorial`, as shown in Figure 4.3, the evaluator will be able to compute factorials.

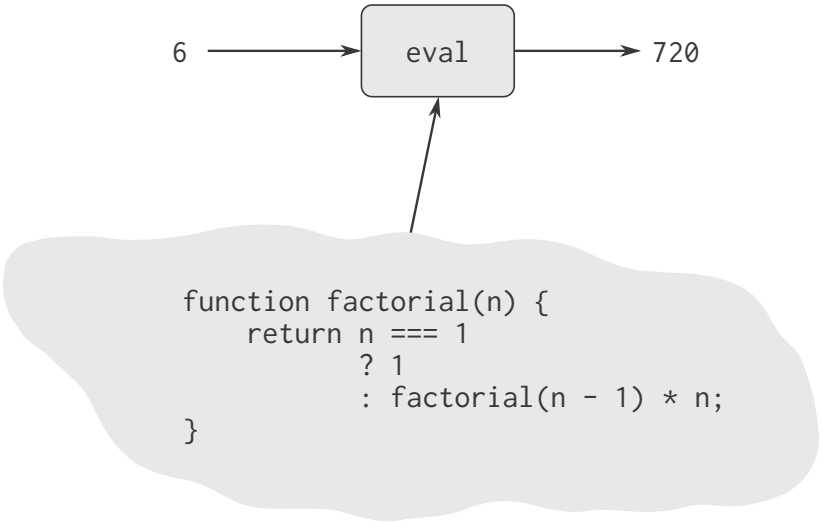


Figure 4.3: The evaluator emulating a factorial machine.

From this perspective, our evaluator is seen to be a *universal machine*. It mimics other machines when these are described as JavaScript programs.[12]

This is striking. Try to imagine an analogous evaluator for electrical circuits. This would be a circuit that takes as input a signal encoding the plans for some other circuit, such as a filter. Given this input, the circuit evaluator would then behave like a filter with the same description. Such a universal electrical circuit is almost unimaginably complex. It is remarkable that the program evaluator is a rather simple program.[13]

[12]The fact that the machines are described in JavaScript is inessential. If we give our evaluator a JavaScript program that behaves as an evaluator for some other language, say C, the JavaScript evaluator will emulate the C evaluator, which in turn can emulate any machine described as a C program. Similarly, writing a JavaScript evaluator in C produces a C program that can execute any JavaScript program. The deep idea here is that any evaluator can emulate any other. Thus, the notion of 'what can in principle be computed' (ignoring practicalities of time and memory required) is independent of the language or the computer, and instead reflects an underlying notion of *computability*. This was first demonstrated in a clear way by Alan M. Turing (1912–1954), whose 1936 paper laid the foundations for theoretical computer science. In the paper, Turing presented a simple computational model—now known as a *Turing machine*—and argued that any 'effective process' can be formulated as a program for such a machine. (This argument is known as the *Church-Turing thesis*.) Turing then implemented a universal machine, i.e., a Turing machine that behaves as an evaluator for Turing-machine programs. He used this framework to demonstrate that there are well-posed problems that cannot be computed by Turing machines (see exercise 4.7), and so by implication cannot be formulated as 'effective processes.' Turing went on to make fundamental contributions to practical computer science as well. For example, he invented the idea of structuring programs using general-purpose subroutines. See Hodges 1983 for a biography of Turing.

[13]Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple function, can emulate programs that are more complex than the evaluator itself. The existence of a universal evaluator machine is a deep and wonderful property of computation. *Recursion theory*, a branch of mathematical logic, is concerned with logical limits of computation. Douglas Hofstadter's beautiful book *Gödel, Escher, Bach* (1979) explores some of these ideas.

Another striking aspect of the evaluator is that it acts as a bridge between the data objects that are manipulated by our programming language and the programming language itself. Imagine that the evaluator program (implemented in JavaScript) is running, and that a user is typing expressions to the evaluator and observing the results. From the perspective of the user, an input expression such as `x * x` is an expression in the programming language, which the evaluator should execute. From the perspective of the evaluator, however, the expression is simply a string or—after parsing—a tagged-object representation that is to be manipulated according to a well-defined set of rules.

That the user's programs are the evaluator's data need not be a source of confusion. In fact, it is sometimes convenient to ignore this distinction, and to give the user the ability to explicitly evaluate a string as a JavaScript expression, using JavaScript's primitive function `eval` that takes as argument a string. It parses the string and—provided that it syntactically correct—evaluates the resulting representation in the environment in which `eval` is applied.[14]

Exercise 4.7

Given a one-argument function `p` and an object `a`, `p` is said to 'halt' on `a` if evaluating the expression `p(a)` returns a value (as opposed to terminating with an error message or running forever). Show that it is impossible to write a function `halts` that correctly determines whether `p` halts on `a` for any function `p` and object `a`. Use the following reasoning: If you had such a function `halts`, you could implement the following program:

```
function run_forever() {
    return run_forever();
}
function try(p) {
    return halts(p, p)
           ? run_forever();
           : "halted";
}
```

Now consider evaluating the expression **`try(try)`** and show that any possible outcome (either halting or running forever) violates the intended behavior of `halts`.[15]

[14] Warning: This `eval` primitive is not identical to the `evaluate` function we implemented in section 4.1.1, because it uses *actual* JavaScript environments rather than the sample environment structures we built in section 4.1.3. These actual environments cannot be manipulated by the user as ordinary lists; they must be accessed via `eval` or other special operations. Similarly, the `apply` primitive we saw in section 2.2.3 is not identical to the metacircular `apply`, because it uses actual JavaScript functions rather than the function objects we constructed in sections 4.1.3 and 4.1.4.

[15] Although we stipulated that `halts` is given a function object, notice that this reasoning still applies even if `halts` can gain access to the function's text and its environment. This is Turing's celebrated *Halting Theorem*, which gave the first clear example of a *non-computable* problem, i.e., a well-posed task that cannot be carried out as a computational function.

4.1.6 Internal Declarations

Our environment model of evaluation and our metacircular evaluator execute declarations in sequence, extending the environment frame one declaration at a time. This is particularly convenient for interactive program development, in which the programmer needs to freely mix the application of functions with the declaration of new functions. However, if we think carefully about the internal declarations used to implement block structure (introduced in section **??**), we will find that name-by-name extension of the environment may not be the best way to declare local names.

Consider a function with internal declarations, such as

```
function f(x) {
    function is_even(n) {
        return n === 0
               ? true
               : is_odd(n - 1);
    }
    function is_odd(n) {
        return n === 0
               ? false
               : is_even(n - 1);
    }
    // rest of body of f
}
```

Our intention here is that the name `is_odd` in the body of the function `is_even` should refer to the function `is_odd` that is declared after `is_even`. The scope of the name `is_odd` is the entire body of `f`, not just the portion of the body of `f` starting at the point where the declaration of `is_odd` occurs. Indeed, when we consider that `is_odd` is itself defined in terms of `is_even`—so that `is_even` and `is_odd` are mutually recursive functions—we see that the only satisfactory interpretation of the two declarations is to regard them as if the names `is_even` and `is_odd` were being added to the environment simultaneously. More generally, in block structure, the scope of a local name is the entire function body in which the declaration is evaluated.

As it happens, our interpreter will evaluate calls to `f` correctly, but for an 'accidental' reason: Since the declarations of the internal functions come first, no calls to these functions will be evaluated until all of them have been declared. Hence, `is_odd` will have been declared by the time `is_even` is executed. In fact, our sequential evaluation mechanism will give the same result as a mechanism that directly implements simultaneous declaration for any function in which the internal declarations come first in a body and evaluation of the value expressions for the declared names doesn't actually use any of the declared names. (For an example of a function that doesn't obey these restrictions, so that sequential declaration isn't equivalent to

simultaneous declaration, see exercise 4.11.)[16]

There is, however, a simple way to treat declarations so that internally declared names have truly simultaneous scope—just create all local names that will be in the current environment before evaluating any of the value expressions. One way to do this is by a syntax transformation on function definition expressions.[17] Before evaluating the body of a function definition expression, we 'scan out' and eliminate all the internal declarations in the body. The internally declared names will be created with a function definition and then set to their values by assignment. In the following, we shall focus on variable declarations using **`let`**; constant declarations using **`const`** and **`function`** can be handled similarly. For example, the function definition

```
(vars) => {
    let u = e1;
    let v = e2;
    statement
}
```

would be transformed into

```
( vars ) => {
    return ( (u, v) => {
                u = e1;
                v = e2;
                statement
            })("*unassigned*", "*unassigned*");
}
```

where `"*unassigned*"` is a special symbol that causes looking up a name to signal an error if an attempt is made to use the value of the not-yet-assigned name.

An alternative strategy for scanning out internal declarations is shown in exercise 4.10. Unlike the transformation shown above, this enforces the restriction that the declared names' values can be evaluated without using any of the names' values.

Exercise 4.8

In this exercise we implement the method just described for interpreting internal definitions.

a. Change `lookup_name_value` (section 4.1.3) to signal an error if the value it finds is the

[16] Wanting programs to not depend on this evaluation mechanism is the reason for the 'management is not responsible' remark in footnote **??** of chapter 1. The designers of JavaScript chose to resolve this issue by moving all internal function declarations to the beginning of the function body, and thus the discussion might seem moot. However, this mechanism is only applied to function declarations and not to **`const`** declarations.

[17] We can view function declaration statements as a combination of constant declaration statements and function definition expressions, as explained in section **??**, and thus the same technique applies to function declaration statements.

string `"*unassigned*"`.

b. Write a function `scan_out_let` that takes a function body and returns an equivalent one that has no internal definitions, by making the transformation described above.

c. Install `scan_out_let` in the interpreter, either in `make_function` or in `function_body` (see section 4.1.3). Which place is better? Why?

Exercise 4.9

Draw diagrams of the environment in effect when evaluating the *statement* in the function in the text, comparing how this will be structured when declarations are interpreted sequentially with how it will be structured if declarations are scanned out as described. Why is there an extra frame in the transformed program? Explain why this difference in environment structure can never make a difference in the behavior of a correct program. Design a way to make the interpreter implement the 'simultaneous' scope rule for internal declarations without constructing the extra frame.

Exercise 4.10

Consider an alternative strategy for scanning out declarations that translates the example in the text to

```
( vars ) => {
    return ( (u, v) => {
                return ( (a, b) => {
                            u = a;
                            v = b;
                            statement
                        })(e1, e2);
            })("*unassigned*", "*unassigned*");
}
```

Here `a` and `b` are meant to represent new variable names, created by the interpreter, that do not appear in the user's program. Consider the `solve` function from section 3.5.4:

```
function solve(f, y0, dt) {
    const y = integral( () => dy, y0, dt);
    const dy = stream_map(f, y);
    return y;
}
```

Will this function work if internal definitions are scanned out as shown in this exercise? What if they are scanned out as shown in the text? Explain.

Exercise 4.11

Ben Bitdiddle, Alyssa P. Hacker, and Eva Lu Ator are arguing about the desired result of evaluating the expression

```
let a = 1;
function f(x) {
    let b = a + x;
    let a = 5;
    return a + b;
}
f(10);
```

Ben asserts that the result should be obtained using the sequential rule for **let**: b is declared to be 11, then a is declared to be 5, so the result is 16. Alyssa objects that mutual recursion requires the simultaneous scope rule for internal function declarations, and that it is unreasonable to treat function names differently from other names. Thus, she argues for the mechanism implemented in exercise 4.8. This would lead to a being unassigned at the time that the value for b is to be computed. Hence, in Alyssa's view the function should produce an error. Eva has a third opinion. She says that if the declarations of a and b are truly meant to be simultaneous, then the value 5 for a should be used in evaluating b. Hence, in Eva's view a should be 5, b should be 15, and the result should be 20. Which (if any) of these viewpoints do you support? Can you devise a way to implement internal declarations so that they behave as Eva prefers?[18]

Exercise 4.12

For recursion, we currently make use of the fact that the scope of a constant declaration is the surrounding block. An occurrence of the function name in its body can refer to the function, because it lies in the scope of the **const** that declares the name. Louis Reasoner thinks that there ought to be a way to specify recursive functions without using **const**, **let** or **function**. Amazingly, Louis's intuition is correct. It is indeed possible to specify recursive functions without using **const** or **let**, **function**, although the method for accomplishing this is much more subtle than Louis imagined. The following expression computes 10 factorial by applying a recursive factorial function:[19]

[18]The designers of JavaScript support Alyssa on the following grounds: Eva is in principle correct—the definitions should be regarded as simultaneous. But it seems difficult to implement a general, efficient mechanism that does what Eva requires. In the absence of such a mechanism, it is better to generate an error in the difficult cases of simultaneous definitions (Alyssa's notion) than to produce an incorrect answer (as Ben would have it).

[19]This example illustrates a programming trick for formulating recursive functions without using **const**, **let** or **function**. The most general trick of this sort is the *Y operator*, which can be used to give a 'pure λ-calculus' implementation of recursion. (See Stoy 1977 for details on the lambda calculus, and Gabriel 1988 for an exposition of the *Y* operator in Scheme.)

```
(n => (fact => fact(fact, n))
      ( (ft, k) => k === 1
                   ? 1
                   : k * ft(ft, k - 1)
      )
)
(10);
```

a. Check (by evaluating the expression) that this really does compute factorials. Devise an analogous expression for computing Fibonacci numbers.

b. Consider the following function, which includes mutually recursive internal definitions:

```
function f(x) {
    function is_even(n) {
        return n === 0
               ? true
               : is_odd(n - 1);
    }
    function is_odd(n) {
        return n === 0
               ? false
               : is_even(n - 1);
    }
    return is_even(x);
}
```

Fill in the missing expressions to complete an alternative definition of `f`, which uses neither **`const`** nor **`let`** nor internal function declarations:

```
function f(x) {
    return (
        (is_even, is_odd) =>
           is_even(is_even, is_odd, x)
      )
      ( (ev, od, n) =>
           n === 0 ? true : od(??, ??, ??),
        (ev, od, n) =>
           n === 0 ? false : ev(??, ??, ??)
      );
}
```

4.1.7 Separating Syntactic Analysis from Execution

Note: this section is a work in progress!

The evaluator implemented above is simple, but it is very inefficient, because the syntactic analysis of expressions is interleaved with their execution. Thus if a program is executed many times, its syntax is analyzed many times. Consider, for example, evaluating `factorial(4)` using the following definition of `factorial`:

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

Each time `factorial` is called, the evaluator must determine that the body is a conditional expression and extract the predicate. Only then can it evaluate the predicate and dispatch on its value. Each time it evaluates the expression `factorial(n - 1) * n`, or the subexpressions `factorial(n - 1)` and `n - 1`, the evaluator must perform the case analysis in `evaluate` to determine that the expression is an application, and must extract its operator and operands. This analysis is expensive. Performing it repeatedly is wasteful.

We can transform the evaluator to be significantly more efficient by arranging things so that syntactic analysis is performed only once.[20] We split `evaluate`, which takes an expression and an environment, into two parts. The function `analyze` takes only the expression. It performs the syntactic analysis and returns a new function, the *execution function*, that encapsulates the work to be done in executing the analyzed expression. The execution function takes an environment as its argument and completes the evaluation. This saves work because `analyze` will be called only once on an expression, while the execution function may be called many times.

With the separation into analysis and execution, `evaluate` now becomes

```
function evaluate(exp, env) {
    return (analyze(exp))(env);
}
```

The result of calling `analyze` is the execution function to be applied to the environment. The `analyze` function is the same case analysis as performed by the original `eval` of section 4.1.1, except that the functions to which we dispatch perform only analysis, not full evaluation:

```
function analyze(stmt) {
    return is_self_evaluating(stmt)
```

[20] This technique is an integral part of the compilation process, which we shall discuss in chapter 5. Jonathan Rees wrote a Scheme interpreter like this in about 1982 for the T project (Rees and Adams 1982). Marc Feeley 1986 (see also Feeley and Lapalme 1987) independently invented this technique in his master's thesis.

```
            ? analyze_self_evaluating(stmt)
          : is_name(stmt)
            ? analyze_name(stmt)
          : is_constant_declaration(stmt)
            ? analyze_constant_declaration(stmt)
          : is_variable_declaration(stmt)
            ? analyze_variable_declaration(stmt)
          : is_assignment(stmt)
            ? analyze_assignment(stmt)
          : is_conditional_expression(stmt)
            ? analyze_conditional_expression(stmt)
          : is_function_definition(stmt)
            ? analyze_function_definition(stmt)
          : is_sequence(stmt)
            ? analyze_sequence(sequence_actions(stmt))
          : is_block(stmt)
            ? analyze_block(block_body(stmt))
          : is_return_statement(stmt)
            ? analyze_return_statement(stmt)
          : is_application(stmt)
            ? analyze_application(stmt)
          : error(stmt, "Unknown statement type in analyze");
}
```

Here is the simplest syntactic analysis function, which handles self-evaluating expressions. It returns an execution function that ignores its environment argument and just returns the expression:

```
function analyze_self_evaluating(stmt) {
    return env => stmt;
}
```

Looking up the value of a name must still be done in the execution phase, since this depends upon knowing the environment.[21]

```
function analyze_name(stmt) {
    return env => lookup_name_value(
                      name_of_name(stmt, env));
}
```

The function `analyze_assignment` also must defer actually setting the variable until the execution, when the environment has been supplied. However, the fact that the The function `assignment_value` expression can be analyzed (recursively) during analysis is a major gain in efficiency, because the The function `assignment_value` expression will now be analyzed only once. The same holds true for constant and variable declarations.

```
function analyze_assignment(stmt) {
```

[21] There is, however, an important part of the variable search that *can* be done as part of the syntactic analysis.

```
    const variable = assignment_variable(stmt);
    const vfun = analyze(assignment_value(stmt));
    return env => {
               set_variable_value(variable,
                   vfun(env), env);
               return "ok";
           };
}
function analyze_variable_declaration(stmt) {
    const name =
        variable_declaration_name(stmt);
    const vfun =
        variable_declaration_value(stmt);
    return env => {
               declare_variable(name,
                                vfun(env),
                                env);
               return "ok";
           };
}
function analyze_constant_declaration(stmt) {
    const name =
        constant_declaration_name(stmt);
    const vfun =
        constant_declaration_value(stmt);
    return env => {
               declare_constant(name,
                                vfun(env),
                                env);
               return "ok";
           };
}
```

For conditional expressions, we extract and analyze the predicate, consequent, and alternative at analysis time.

```
function analyze_conditional_expression(stmt) {
    const pfun = analyze(cond_expr_pred(stmt));
    const cfun = analyze(cond_expr_cons(stmt));
    const afun = analyze(cond_expr_alt(stmt));
    return env => is_true(pfun(env))
                  ? cfun(env)
                  : afun(env);
}
```

Analyzing a function definition expression also achieves a major gain in efficiency: We analyze the function body only once, even though functions resulting from evaluation of the function definition may be applied many times.

```
function analyze_function_definition(stmt) {
    const vars =
        function_definition_parameters(stmt);
    const bfun =
        analyze(function_definition_body(stmt);
    return env =>
        make_function(vars, bfun, env);
}
```

Analysis of a sequence of statements is more involved.[22] Each statement in the sequence is analyzed, yielding an execution function. These execution functions are combined to produce an execution function that takes an environment as argument and sequentially calls each individual execution function with the environment as argument.

```
function analyze_sequence(stmts) {
    function sequentially(fun1, fun2) {
        return env => {
                   fun1(env);
                   fun2(env);
               };
    }
    function loop(first_fun, rest_funs) {
        return is_null(rest_funs)
               ? first_fun
               : loop(sequentially(first_fun,
                                   head(rest_funs)),
                      tail(rest_funs));
    }
    const funs = map(analyze, stmts);
    return is_null(funs)
           ? env => undefined
           : loop(head(funs), tail(funs));
}
```

To analyze an application, we analyze the operator and operands and construct an execution function that calls the operator execution function (to obtain the actual function to be applied) and the operand execution functions (to obtain the actual arguments). We then pass these to `execute_application`, which is the analog of `apply` in section 4.1.1. The function `execute_application` differs from `apply` in that the function body for a compound function has already been analyzed, so there is no need to do further analysis. Instead, we just call the execution function for the body on the extended environment.

```
function analyze_application(stmt) {
    const ffun = analyze(operator(stmt));
    const afuns = map(analyze, operands(stmt));
    return env =>
```

[22] See exercise 4.14 for some insight into the processing of sequences.

```
            execute_application(ffun(env),
                map(afun => afun(env), afuns));
}
function execute_application(fun, args) {
    return is_primitive_function(fun)
           ? apply_primitive_function(fun, args)
           : is_compound_function(fun)
             ? (function_body(fun))
               (extend_environment(
                    function_parameters(fun),
                    args,
                    function_environment(fun)))
             : error(fun, "unknown function type in " +
                          "execute_application");
}
```

Our new evaluator uses the same data structures, syntax functions, and run-time support functions as in sections 4.1.2, 4.1.3, and 4.1.4.

Exercise 4.13

Extend the evaluator in this section to support conditional expressions.

Exercise 4.14

Alyssa P. Hacker doesn't understand why analyze_sequence needs to be so complicated. All the other analysis functions are straightforward transformations of the corresponding evaluation functions (or eval clauses) in section 4.1.1. She expected analyze_sequence to look like this:

```
function analyze_sequence(stmts) {
    function execute_sequence(funs, env) {
        if (is_null(tail(funs))) {
            return head(funs)(env);
        } else {
            head(funs)(env);
            execute_sequence(tail(funs),
                             env);
        }
    }
    const funs = map(analyze, stmts);
    return is_null(funs)
           ? env => undefined
           : env => execute_sequence(funs,
                                     env);
}
```

Eva Lu Ator explains to Alyssa that the version in the text does more of the work of evaluating a sequence at analysis time. Alyssa's `sequence_execution` function, rather than having the calls to the individual execution functions built in, loops through the functions in order to call them: In effect, although the individual expressions in the sequence have been analyzed, the sequence itself has not been. Compare the two versions of `sequence_execution`. For example, consider the common case (typical of function bodies) where the sequence has just one expression. What work will the execution function produced by Alyssa's program do? What about the execution function produced by the program in the text above? How do the two versions compare for a sequence with two expressions?

Exercise 4.15

Design and carry out some experiments to compare the speed of the original metacircular evaluator with the version in this section. Use your results to estimate the fraction of time that is spent in analysis versus execution for various functions.

4.2 Lazy Evaluation

Note: this section is a work in progress!

Now that we have an evaluator expressed as a JavaScript program, we can experiment with alternative choices in language design simply by modifying the evaluator. Indeed, new languages are often invented by first writing an evaluator that embeds the new language within an existing high-level language. For example, if we wish to discuss some aspect of a proposed modification to JavaScript with another member of the JavaScript community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications. Not only does the high-level implementation base make it easier to test and debug the evaluator; in addition, the embedding enables the designer to snarf[23] features from the underlying language, just as our embedded JavaScript evaluator uses primitives and control structure from the underlying JavaScript. Only later (if ever) need the designer go to the trouble of building a complete implementation in a low-level language or in hardware. In this section and the next we explore some variations on JavaScript that provide significant additional expressive power.

[23] Snarf: 'To grab, especially a large document or file for the purpose of using it either with or without the owner's permission.' Snarf down: 'To snarf, sometimes with the connotation of absorbing, processing, or understanding.' (These definitions were snarfed from Steele et al. 1983. See also Raymond 1993.)

4.2.1 Normal Order and Applicative Order

In section **??**, where we began our discussion of models of evaluation, we noted that JavaScript is an *applicative-order* language, namely, that all the arguments to JavaScript functions are evaluated when the function is applied. In contrast, *normal-order* languages delay evaluation of function arguments until the actual argument values are needed. Delaying evaluation of function arguments until the last possible moment (e.g., until they are required by a primitive operation) is called *lazy evaluation.*[24] Consider the function

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

Evaluating `try_me(0, head(null));` generates an error in JavaScript. With lazy evaluation, there would be no error. Evaluating the expression would return 1, because the argument `head(null)` would never be evaluated.

An example that exploits lazy evaluation is the definition of a function `unless` that can be used in expressions such as

```
unless(xs === null,
       head(xs),
       display("error: xs should not be null"));
```

This won't work in an applicative-order language because both the usual value and the exceptional value will be evaluated before `unless` is called (compare exercise **??**). An advantage of lazy evaluation is that some functions, such as `unless`, can do useful computation even if evaluation of some of their arguments would produce errors or would not terminate.

If the body of a function is entered before an argument has been evaluated we say that the function is *non-strict* in that argument. If the argument is evaluated before the body of the function is entered we say that the function is *strict* in that argument.[25]

In a purely applicative-order language, all functions are strict in each argument. In a purely normal-order language, all compound functions are non-strict in each argument, and primitive functions may be either strict or non-strict. There are also languages (see exercise 4.22) that give programmers detailed control over the strictness of the functions they define.

A striking example of a function that can usefully be made non-strict is `pair` (or, in general,

[24] The difference between the 'lazy' terminology and the 'normal-order' terminology is somewhat fuzzy. Generally, 'lazy' refers to the mechanisms of particular evaluators, while 'normal-order' refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably.

[25] The 'strict' versus 'non-strict' terminology means essentially the same thing as 'applicative-order' versus 'normal-order,' except that it refers to individual functions and arguments rather than to the language as a whole. At a conference on programming languages you might hear someone say, 'The normal-order language Hassle has certain strict primitives. Other functions take their arguments by lazy evaluation.'

almost any constructor for data structures). One can do useful computation, combining elements to form data structures and operating on the resulting data structures, even if the values of the elements are not known. It makes perfect sense, for instance, to compute the length of a list without knowing the values of the individual elements in the list. We will exploit this idea in section 4.2.3 to implement the streams of chapter 3 as lists formed of non-strict `pair` pairs.

Exercise 4.16

Suppose that (in ordinary applicative-order JavaScript) we define `unless` as shown above and then define `factorial` in terms of `unless` as

```
function factorial(n) {
    return unless(n === 1,
                  n * factorial(n - 1),
                  1);
}
```

What happens if we attempt to evaluate `factorial(5)`? Will our definitions work in a normal-order language?

Exercise 4.17

Ben Bitdiddle and Alyssa P. Hacker disagree over the importance of lazy evaluation for implementing things such as `unless`. Ben points out that it's possible to implement `unless` in applicative order as a new kind of expression, akin to conditional expressions. Alyssa counters that, if one did that, `unless` would be merely syntax, not a function that could be used in conjunction with higher-order functions. Fill in the details on both sides of the argument. Show how to define the evaluation of `unless` as a new kind of expression (as we defined the evaluation of conditional expressions in section **??**), and give an example of a situation where it might be useful to have `unless` available as a function, rather than as a new expression syntax.

4.2.2 An Interpreter with Lazy Evaluation

In this section we will implement a normal-order language that is the same as JavaScript except that compound functions are non-strict in each argument. Primitive functions will still be strict. It is not difficult to modify the evaluator of section 4.1.1 so that the language it interprets behaves this way. Almost all the required changes center around function application.

The basic idea is that, when applying a function, the interpreter must determine which arguments are to be evaluated and which are to be delayed. The delayed arguments are not

evaluated; instead, they are transformed into objects called *thunk*s.[26]

The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application. Thus, the thunk must contain the argument expression and the environment in which the function application is being evaluated.

The process of evaluating the expression in a thunk is called *forcing*.[27]

In general, a thunk will be forced only when its value is needed: when it is passed to a primitive function that will use the value of the thunk; when it is the value of a predicate of a conditional; and when it is the value of an operator that is about to be applied as a function. One design choice we have available is whether or not to *memoize* thunks, as we did with delayed objects in section 3.5.1. With memoization, the first time a thunk is forced, it stores the value that is computed. Subsequent forcings simply return the stored value without repeating the computation. We'll make our interpreter memoize, because this is more efficient for many applications. There are tricky considerations here, however.[28]

Modifying the evaluator

The main difference between the lazy evaluator and the one in section 4.1 is in the handling of function applications in `evaluate` and `apply`.

The `is_application` clause of `evaluate` becomes

```
is_application(exp)
? apply(actual_value(operator(exp), env),
        operands(exp),
        env)
```

This is almost the same as the `is_application` clause of `evaluate` in section 4.1.1. For lazy evaluation, however, we call `apply` with the operand expressions, rather than the arguments produced by evaluating them. Since we will need the environment to construct thunks if the arguments are to be delayed, we must pass this as well. We still evaluate the operator, because

[26] The word *thunk* was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of ('thinking about') the expression could be done at compile time; thus, at run time, the expression would already have been 'thunk' about (Ingerman et al. 1960).

[27] This is analogous to the use of `force` on the delayed objects that were introduced in chapter 3 to represent streams. The critical difference between what we are doing here and what we did in chapter 3 is that we are building delaying and forcing into the evaluator, and thus making this uniform and automatic throughout the language.

[28] Lazy evaluation combined with memoization is sometimes referred to as *call-by-need* argument passing, in contrast to *call-by-name* argument passing. (Call-by-name, introduced in Algol 60, is similar to non-memoized lazy evaluation.) As language designers, we can build our evaluator to memoize, not to memoize, or leave this an option for programmers (exercise 4.22). As you might expect from chapter 3, these choices raise issues that become both subtle and confusing in the presence of assignments. (See exercises 4.18 and 4.20.) An excellent article by Clinger (1982) attempts to clarify the multiple dimensions of confusion that arise here.

apply needs the actual function to be applied in order to dispatch on its type (primitive versus compound) and apply it.

Whenever we need the actual value of an expression, we use

```
function actual_value(exp, env) {
    return force_it(evaluate(exp, env));
}
```

instead of just evaluate, so that if the expression's value is a thunk, it will be forced.

Our new version of apply is also almost the same as the version in section 4.1.1. The difference is that evaluate has passed in unevaluated operand expressions: For primitive functions (which are strict), we evaluate all the arguments before applying the primitive; for compound functions (which are non-strict) we delay all the arguments before applying the function.

```
function apply(fun, args) {
   if (is_primitive_function(fun)) {
       return apply_primitive_function(
                  fun,  // following line changed
                  list_of_arg_values(args, env));
   } else if (is_compound_function(fun)) {
      const result =
         evaluate(function_body(fun),
                  extend_environment(
                      function_parameters(fun),
                        // following line changed
                      list_of_delayed_args(args,
                                           env),
                      function_environment(fun)));
      if (is_return_value(result)) {
         return return_value_content(result);
      } else {
          return undefined;
      }
   } else {
      Error("Unknown function type in apply",
            fun);
   }
}
```

The functions that process the arguments are just like list_of_values from section 4.1.1, except that list_of_delayed_args delays the arguments instead of evaluating them, and list_of_arg_values uses actual_value instead of evaluate:

```
function list_of_arg_values(exps, env) {
    return no_operands(exps)
        ? null
        : pair(actual_value(first_operand(exps),
```

```
                              env),
                list_of_arg_values(rest_operands(exps),
                                   env));
}
function list_of_delayed_args(exps, env) {
    return no_operands(exps)
        ? null
        : pair(delay_it(first_operand(exps), env),
               list_of_delayed_args(
                   rest_operands(exps), env));
}
```

The other place we must change the evaluator is in the handling of **`if`**, where we must use `actual-value` instead of `eval` to get the value of the predicate expression before testing whether it is true or false:

```
function eval_conditional_expression(exp, env) {
    return is_true(actual_value(cond_expr_pred(exp),
                                env))
           ? evaluate(cond_expr_cons(exp), env)
           : evaluate(cond_expr_alt(exp), env);
}
```

Finally, we must change the `eval_toplevel` function (section 4.1.4) to use `actual_value` instead of `evaluate`, so that if a delayed value is propagated back to the evaluator it will be forced before being printed.

```
function eval_toplevel(stmt) {
   const value = actual_value(stmt, the_global_environment);
   if (is_return_value(value)) {
       error("return not allowed " +
             "outside of function definitions");
   } else {
       return value;
   }
}
```

With these changes made, we can start the evaluator and test it. The successful evaluation of the **`try`** expression discussed in section 4.2.1 indicates that the interpreter is performing lazy evaluation:

```
eval_toplevel(parse(
   "function try(a, b) {        " +
   "    return a === 0 ? 1 : b; " +
   "}                           " +
   "try(0, head(null));         "   ));
```

Representing thunks

Our evaluator must arrange to create thunks when functions are applied to arguments and to force these thunks later. A thunk must package an expression together with the environment, so that the argument can be produced later. To force the thunk, we simply extract the expression and environment from the thunk and evaluate the expression in the environment. We use `actual_value` rather than `evaluate` so that in case the value of the expression is itself a thunk, we will force that, and so on, until we reach something that is not a thunk:

```
function force_it(obj) {
     return is_thunk(obj)
            ? actual_value(thunk_exp(obj), thunk_env(obj))
            : obj;
}
```

One easy way to package an expression with an environment is to make a list containing the expression and the environment. Thus, we create a thunk as follows:

```
function delay_it(exp, env) {
    return list("thunk", exp, env);
}
function is_thunk(obj) {
    return is_tagged_list(obj, "thunk");
}
function thunk_exp(thunk) {
    return head(tail(thunk));
}
function thunk_env(thunk) {
    return head(tail(tail(thunk)));
}
```

Actually, what we want for our interpreter is not quite this, but rather thunks that have been memoized. When a thunk is forced, we will turn it into an evaluated thunk by replacing the stored expression with its value and changing the `thunk` tag so that it can be recognized as already evaluated.[29]

```
function is_evaluated_thunk(obj) {
    return is_tagged_list(obj, "evaluated_thunk");
}
function thunk_value(evaluated_thunk) {
    return head(tail(evaluated_thunk));
}
```

[29] Notice that we also erase the `env` from the thunk once the expression's value has been computed. This makes no difference in the values returned by the interpreter. It does help save space, however, because removing the reference from the thunk to the `env` once it is no longer needed allows this structure to be *garbage-collected* and its space recycled Similarly, we could have allowed unneeded environments in the memoized delayed objects of section 3.5.1 to be garbage-collected, by having `memo_fun` do something like `fun = null;` to discard the function `fun` (which includes the environment in which the `delay` was evaluated) after storing its value.

```
function force_it(obj) {
    if (is_thunk(obj)) {
        const result = actual_value(
                           thunk_exp(obj),
                           thunk_env(obj));
        set_head(obj, "evaluated_thunk");
        // replace exp with its value
        set_head(tail(obj), result);
        // forget unneeded env
        set_tail(tail(obj), null);
        return result;
    } else if(is_evaluated_thunk(obj)) {
        return thunk_value(obj);
    } else {
        return obj;
    }
}
```

Notice that the same `delay_it` function works both with and without memoization.

Exercise 4.18

Suppose we type in the following definitions to the lazy evaluator:

```
let count = 0;
function id(x) {
    count = count + 1;
    return x;
}
```

Give the missing values in the following sequence of interactions, and explain your answers.[30]

```
read_eval_print_loop("");
// enter:    <count and id as defined above>
// response: ?
// enter:    const w = id(id(10));
// response: ?
// enter:    count
// response: ?
// enter:    w
// response: ?
// enter:    count
// response: ?
```

Exercise 4.19

[30] This exercise demonstrates that the interaction between lazy evaluation and side effects can be very confusing. This is just what you might expect from the discussion in chapter 3.

The function `evaluate` uses `actual_value` rather than `evaluate` to evaluate the operator before passing it to `apply`, in order to force the value of the operator. Give an example that demonstrates the need for this forcing.

Exercise 4.20

Exhibit a program that you would expect to run much more slowly without memoization than with memoization. Also, consider the following interaction, where the `id` function is defined as in exercise 4.18 and `count` starts at 0:

```
read_eval_print_loop("");
// enter:    <count and id as defined above>
// response: ?
// enter:    function square(x) { return x * x; }
// response: ?
// enter:    square(id(10));
// response: ?
// enter:    count
// response: ?
```

Give the responses both when the evaluator memoizes and when it does not.

Exercise 4.21

Cy D. Fect, a reformed C programmer, is worried that some side effects may never take place, because the lazy evaluator doesn't force the expressions in a sequence. Since the value of an expression in a sequence other than the last one is not used (the expression is there only for its effect, such as assigning to a variable or printing), there can be no subsequent use of this value (e.g., as an argument to a primitive function) that will cause it to be forced. Cy thus thinks that when evaluating sequences, we must force all expressions in the sequence except the final one. He proposes to modify `eval-sequence` from section 4.1.1 to use `actual_value` rather than `evaluate`:

```
function eval_sequence(exps, env) {
    if (is_last_exp(exps)) {
        evaluate(first_exp(exps), env);
    } else {
        actual_value(first_exp(exps), env);
        eval_sequence(rest_exps(exps), env);
    }
}
```

a. Ben Bitdiddle thinks Cy is wrong. He shows Cy the `for_each` function described in exercise 2.7, which gives an important example of a sequence with side effects:

```
function for_each(fun, items) {
    if (is_null(items)){
        return undefined;
    } else {
        fun(head(items));
        for_each(fun, tail(items));
    }
}
```

He claims that the evaluator in the text (with the original `eval_sequence`) handles this correctly:

```
read_eval_print_loop("");
// enter:    <for_each as defined above>
// response: ?
// enter:    for_each(x => display(x), list(57, 321, 88));
// response: 57
//           321
//           88
// response: done
```

Explain why Ben is right about the behavior of `for_each`.

b. Cy agrees that Ben is right about the `for_each` example, but says that that's not the kind of program he was thinking about when he proposed his change to `eval_sequence`. He defines the following two functions in the lazy evaluator:

```
function p1(x) {
    x = pair(x, list(2));
}
function p2(x) {
    function p(e) {
        e;
        return x;
    }
    x = pair(x, list(2));
    return p(x);
}
```

What are the values of `p(1)` and `p2(1)` with the original `eval_sequence`? What would the values be with Cy's proposed change to `eval_sequence`?

c. Cy also points out that changing `eval_sequence` as he proposes does not affect the behavior of the example in part a. Explain why this is true.

d. How do you think sequences ought to be treated in the lazy evaluator? Do you like Cy's approach, the approach in the text, or some other approach?

Exercise 4.22

The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to JavaScript. It might be nicer to implement lazy evaluation as an *upward-compatible extension*, that is, so that ordinary JavaScript programs will work as before. We can do this by extending the syntax of function declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

```
function f(a, b, c, d) {
    parameters("strict", "lazy", "strict", "lazy-memo");
    ...
}
```

would define `f` to be a function of four arguments, where the first and third arguments are evaluated when the function is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary function definitions will produce the same behavior as ordinary JavaScript, while adding the `"lazy-memo"` declaration to each parameter of every compound function will produce the behavior of the lazy evaluator defined in this section. Design and implement the changes required to produce such an extension to JavaScript. You can assume that the special 'function call' `parameters` is always the first statement in the body of a function declaration. You must also arrange for `evaluate` or `apply` to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

4.2.3 Streams as Lazy Lists

In section 3.5.1, we showed how to implement streams as delayed lists. We used a function definition expression to construct a 'promise' to compute the `tail` of a stream, without actually fulfilling that promise until later. We were forced to create streams as a new kind of data object similar but not identical to lists, and this required us to reimplement many ordinary list operations (`map`, `append`, and so on) for use with streams.

With lazy evaluation, streams and lists can be identical, so there is no need for separate list and stream operations. All we need to do is to arrange matters so that `pair` is non-strict. One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives, and to implement `pair` as one of these. An easier way is to recall (section **??**) that there is no fundamental need to implement `pair` as a primitive at all. Instead, we can represent pairs as functions:[31]

[31] This is the functional representation described in exercise **??**. Essentially any functional representation (e.g., a message-passing implementation) would do as well. Notice that we can install these definitions in the lazy evaluator simply by typing them at the driver loop. If we had originally included `pair`, `head`, and `tail` as primitives

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

In terms of these basic operations, the standard definitions of the list operations will work with infinite lists (streams) as well as finite ones, and the stream operations can be implemented as list operations. Here are some examples:

```
function list_ref(items, n) {
    return n === 0
           ? head(items)
           : list_ref(tail(items), n - 1);
}
function map(fun, items) {
    return is_null(items)
           ? null
           : pair(fun(head(items)),
                  map(fun, tail(items)));
}
function scale_list(items, factor) {
    return map(x => x * factor, items);
}
function add_lists(list1, list2) {
    return is_null(list1)
           ? list2
           : is_null(list2)
             ? list1
             : pair(head(list1) + head(list2),
                    add_lists(tail(list1),
                              tail(list2)));
}
const ones = pair(1, ones);
const integers = pair(1, add_lists(ones, integers));

list_ref(integers, 17); // returns 18
```

Note that these lazy lists are even lazier than the streams of chapter 3: The `head` of the list, as well as the `tail`, is delayed.[32] In fact, even accessing the `head` or `tail` of a lazy pair need not force the value of a list element. The value will be forced only when it is really needed—e.g.,

in the global environment, they will be redefined. (Also see exercises 4.24 and 4.25.)

[32] This permits us to create delayed versions of more general kinds of list structures, not just sequences. Hughes 1990 discusses some applications of 'lazy trees.'

for use as the argument of a primitive, or to be printed as an answer.

Lazy pairs also help with the problem that arose with streams in section 3.5.4, where we found that formulating stream models of systems with loops may require us to sprinkle our programs with additional delayed function definitions, beyond the ones required to construct a stream pair. With lazy evaluation, all arguments to functions are delayed uniformly. For instance, we can implement functions to integrate lists and solve differential equations as we originally intended in section 3.5.4:

```
function integral(integrand, initial_value, dt) {
    const int =
        pair(initial_value,
             add_lists(scale_list(integrand, dt),
                       int));
    return int;
}
function solve(f, y0, dt) {
    const y = integral(dy, y0, dt);
    const dy = map(f, y);
    return y;
}
list_ref(solve(x => x, 1, 0.001), 1000);
```

Exercise 4.23

Give some examples that illustrate the difference between the streams of chapter 3 and the 'lazier' lazy lists described in this section. How can you take advantage of this extra laziness?

Exercise 4.24

Ben Bitdiddle tests the lazy list implementation given above by evaluating the expression `head(list(a, b, c));`

To his surprise, this produces an error. After some thought, he realizes that the 'lists' obtained by reading in quoted expressions are different from the lists manipulated by the new definitions of `pair`, `head`, and `tail`. Modify the evaluator's treatment of applications of the primitive function l `list` typed at the driver loop will produce true lazy lists.

Exercise 4.25

Modify the driver loop for the evaluator so that lazy pairs and lists will print in some reasonable way. (What are you going to do about infinite lists?) You may also need to modify the representation of lazy pairs so that the evaluator can identify them in order to print them.

4.3 Nondeterministic Computing

Note: this section is a work in progress!

In this section, we extend the JavaScript evaluator to support a programming paradigm called *nondeterministic computing* by building into the evaluator a facility to support automatic search. This is a much more profound change to the language than the introduction of lazy evaluation in section 4.2.

Nondeterministic computing, like stream processing, is useful for 'generate and test' applications. Consider the task of starting with two lists of positive integers and finding a pair of integers—one from the first list and one from the second list—whose sum is prime. We saw how to handle this with finite sequence operations in section 2.1.3 and with infinite streams in section 3.5.3. Our approach was to generate the sequence of all possible pairs and filter these to select the pairs whose sum is prime. Whether we actually generate the entire sequence of pairs first as in chapter 2, or interleave the generating and filtering as in chapter 3, is immaterial to the essential image of how the computation is organized.

The nondeterministic approach evokes a different image. Imagine simply that we choose (in some way) a number from the first list and a number from the second list and require (using some mechanism) that their sum be prime. This is expressed by following function:

```
function prime_sum_pair(list1, list2) {
    const a = an_element_of(list1);
    const b = an_element_of(list2);
    require(is_prime(a + b);
    return list(a, b);
}
```

It might seem as if this function merely restates the problem, rather than specifying a way to solve it. Nevertheless, this is a legitimate nondeterministic program.[33]

The key idea here is that expressions in a nondeterministic language can have more than one possible value. For instance, `an_element_of` might return any element of the given list. Our nondeterministic program evaluator will work by automatically choosing a possible value and keeping track of the choice. If a subsequent requirement is not met, the evaluator will try a different choice, and it will keep trying new choices until the evaluation succeeds, or until we run out of choices. Just as the lazy evaluator freed the programmer from the details

[33] We assume that we have previously defined a function `is_prime` that tests whether numbers are prime. Even with `is_prime` defined, the `prime_sum_pair` function may look suspiciously like the unhelpful 'pseudo-JavaScript' attempt to define the square-root function, which we described at the beginning of section **??**. In fact, a square-root function along those lines can actually be formulated as a nondeterministic program. By incorporating a search mechanism into the evaluator, we are eroding the distinction between purely declarative descriptions and imperative specifications of how to compute answers. We'll go even farther in this direction in section 4.4.

of how values are delayed and forced, the nondeterministic program evaluator will free the programmer from the details of how choices are made.

It is instructive to contrast the different images of time evoked by nondeterministic evaluation and stream processing. Stream processing uses lazy evaluation to decouple the time when the stream of possible answers is assembled from the time when the actual stream elements are produced. The evaluator supports the illusion that all the possible answers are laid out before us in a timeless sequence. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Some of the possible worlds lead to dead ends, while others have useful values. The nondeterministic program evaluator supports the illusion that time branches, and that our programs have different possible execution histories. When we reach a dead end, we can revisit a previous choice point and proceed along a different branch.

The nondeterministic program evaluator implemented below is called the `amb` evaluator because it is based on a special 'function' called `amb`. We can type the above definition of `prime_sum_pair` at the `amb` evaluator driver loop (along with definitions of `is_prime`, `is_prime`, `an_element_of`, and `require`) and run the function as follows:

```
Amb-Eval input:
prime_sum_pair(list(1, 3, 5, 8), list(20, 35, 110));
Starting a new problem
Amb-Eval value:
[3, [20, null]]
```

The value returned was obtained after the evaluator repeatedly chose elements from each of the lists, until a successful choice was made.

Section 4.3.1 introduces `amb` and explains how it supports nondeterminism through the evaluator's automatic search mechanism. Section 4.3.2 presents examples of nondeterministic programs, and section 4.3.3 gives the details of how to implement the `amb` evaluator by modifying the ordinary Scheme evaluator.

4.3.1 Amb and Search

To extend JavaScript to support nondeterminism, we introduce a new kind of expression called `amb`.[34]

The expression `amb`$(e_1, e_2, \ldots, e_n)$ returns the value of one of the n expressions e_i 'ambiguously.' For example, the expression

[34]The idea of `amb` for nondeterministic programming was first described in 1961 by John McCarthy (see McCarthy 1967). For convenience, we make these expressions look like function applications. This is analogous to the lazy boolean operators, which look like binary operators, but which are treated differently by the evaluator. Applications of `amb` will be treated differently from applications of ordinary primitive functions.

```
list(amb(1, 2, 3), amb("a", "b"))
```

can have six possible values:

```
list(1, "a") list(1, "b") list(2, "a") list(2, "b") list(3, "a") list(3, "b")
```

An application of amb with a single choice produces an ordinary (single) value.

An application of amb with no choices—the expression amb()—is an expression with no acceptable values. Operationally, we can think of amb() as an expression that when evaluated causes the computation to 'fail': The computation aborts and no value is produced. Using this idea, we can express the requirement that a particular predicate expression p must be true as follows:

```
function require(p) {
    return ! p ? amb() : "some ordinary value";
}
```

With amb and require, we can implement the an_element_of function used above:

```
function an_element_of(items) {
    require(! is_null(items));
    return amb(head(items), an_element_of(tail(items));
}
```

An application of an_element_of fails if the list is empty. Otherwise it ambiguously returns either the first element of the list or an element chosen from the rest of the list.

We can also express infinite ranges of choices. The following function potentially returns any integer greater than or equal to some given *n*:

```
function an_integer_starting_from(n) {
    return amb(n, an_integer_starting_from(n + 1));
}
```

This is like the stream function integers_starting_from described in section 3.5.2, but with an important difference: The stream function returns an object that represents the sequence of all integers beginning with *n*, whereas the amb function returns a single integer.[35]

Abstractly, we can imagine that evaluating an amb expression causes time to split into branches, where the computation continues on each branch with one of the possible values of the expression. We say that amb represents a *nondeterministic choice point*. If we had a machine with a sufficient number of processors that could be dynamically allocated, we could implement the search in a straightforward way. Execution would proceed as in a sequential machine, until

[35] In actuality, the distinction between nondeterministically returning a single choice and returning all choices depends somewhat on our point of view. From the perspective of the code that uses the value, the nondeterministic choice returns a single value. From the perspective of the programmer designing the code, the nondeterministic choice potentially returns all possible values, and the computation branches so that each value is investigated separately.

an amb expression is encountered. At this point, more processors would be allocated and initialized to continue all of the parallel executions implied by the choice. Each processor would proceed sequentially as if it were the only choice, until it either terminates by encountering a failure, or it further subdivides, or it finishes.[36]

On the other hand, if we have a machine that can execute only one process (or a few concurrent processes), we must consider the alternatives sequentially. One could imagine modifying an evaluator to pick at random a branch to follow whenever it encounters a choice point. Random choice, however, can easily lead to failing values. We might try running the evaluator over and over, making random choices and hoping to find a non-failing value, but it is better to *systematically search* all possible execution paths. The amb evaluator that we will develop and work with in this section implements a systematic search as follows: When the evaluator encounters an application of amb, it initially selects the first alternative. This selection may itself lead to a further choice. The evaluator will always initially choose the first alternative at each choice point. If a choice results in a failure, then the evaluator automagically[37] *backtracks* to the most recent choice point and tries the next alternative. If it runs out of alternatives at any choice point, the evaluator will back up to the previous choice point and resume from there. This process leads to a search strategy known as *depth-first search* or *chronological backtracking*.[38]

[36] One might object that this is a hopelessly inefficient mechanism. It might require millions of processors to solve some easily stated problem this way, and most of the time most of those processors would be idle. This objection should be taken in the context of history. Memory used to be considered just such an expensive commodity. In 1964 a megabyte of RAM cost about $400,000. Now every personal computer has many megabytes of RAM, and most of the time most of that RAM is unused. It is hard to underestimate the cost of mass-produced electronics.

[37] Automagically: 'Automatically, but in a way which, for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), the speaker doesn't feel like explaining.' (Steele 1983, Raymond 1993)

[38] The integration of automatic search strategies into programming languages has had a long and checkered history. The first suggestions that nondeterministic algorithms might be elegantly encoded in a programming language with search and automatic backtracking came from Robert Floyd (1967). Carl Hewitt (1969) invented a programming language called Planner that explicitly supported automatic chronological backtracking, providing for a built-in depth-first search strategy. Sussman, Winograd, and Charniak (1971) implemented a subset of this language, called MicroPlanner, which was used to support work in problem solving and robot planning. Similar ideas, arising from logic and theorem proving, led to the genesis in Edinburgh and Marseille of the elegant language Prolog (which we will discuss in section 4.4). After sufficient frustration with automatic search, McDermott and Sussman (1972) developed a language called Conniver, which included mechanisms for placing the search strategy under programmer control. This proved unwieldy, however, and Sussman and Stallman (1975) found a more tractable approach while investigating methods of symbolic analysis for electrical circuits. They developed a non-chronological backtracking scheme that was based on tracing out the logical dependencies connecting facts, a technique that has come to be known as *dependency-directed backtracking*. Although their method was complex, it produced reasonably efficient programs because it did little redundant search. Doyle 1979 and McAllester (McAllester 1978, McAllester 1980) generalized and clarified the methods of Stallman and Sussman, developing a new paradigm for formulating search that is now called *truth maintenance*. Modern problem-solving systems all use some form of truth-maintenance system as a substrate. See Forbus and deKleer 1993 for a discussion of elegant ways to build truth-maintenance systems and applications using truth maintenance. Zabih, McAllester, and Chapman 1987 describes a nondeterministic extension to Scheme that is based on amb; it is similar to the interpreter described in this section, but more sophisticated, because it uses dependency-directed backtracking rather than chronological backtracking. Winston 1992 gives an introduction to both kinds of backtracking.

Driver loop

The driver loop for the amb evaluator has some unusual properties. It reads an expression and prints the value of the first non-failing execution, as in the example shown above. If we want to see the value of the next successful execution, we can ask the interpreter to backtrack and attempt to generate a second non-failing execution. This is signaled by typing **try**-again. If any other input except **try**-again is given, the interpreter will start a new problem, discarding the unexplored alternatives in the previous problem. Here is a sample interaction:

```
// Amb-Eval input:
prime_sum_pair(list(1, 3, 5, 8), list(20, 35, 110));
// Starting a new problem
// Amb-Eval value:
[3, [20, null]]

// Amb-Eval input:
try-again
// Amb-Eval value:
[3, [110, null]]

// Amb-Eval input:
try-again
// Amb-Eval value:
[8, [35, null]]

// Amb-Eval input:
try-again
// There are no more values of
prime_sum_pair([1, [3, [5, [8, null]]]], [20, [35, [110, null]]])

// Amb-Eval input:
prime_sum_pair(list(19, 27, 30), list(11, 36, 58));
// Starting a new problem
// Amb-Eval value:
[30, [11, null]]
```

Exercise 4.26

Write a function an_integer_between that returns an integer between two given bounds. This can be used to implement a function that finds Pythagorean triples, i.e., triples of integers (i, j, k) between the given bounds such that $i \leq j$ and $i^2 + j^2 = k^2$, as follows:

```
function a_pythogorean_triple_between(low, high) {
    const i = an_integer_between(low, high);
    const j = an_integer_between(i, high);
    const k = an_integer_between(j, high);
```

```
    require(i * i + j * j === k * k);
    return list(i, j, k);
}
```

Exercise 4.27

exercise 3.69 discussed how to generate the stream of *all* Pythagorean triples, with no upper bound on the size of the integers to be searched. Explain why simply replacing `an_integer_between` by `an_integer_starting_from` in the function in exercise 4.26 is not an adequate way to generate arbitrary Pythagorean triples. Write a function that actually will accomplish this. (That is, write a function for which repeatedly typing would in principle eventually generate all Pythagorean triples.)

Exercise 4.28

Ben Bitdiddle claims that the following method for generating Pythagorean triples is more efficient than the one in exercise 4.26. Is he correct? (Hint: Consider the number of possibilities that must be explored.)

```
function a_pythagorean_triple_between(low, high) {
    const i = an_integer_between(low, high);
    const hsq = high * high;
    const j = an_integer_between(i, high);
    const ksq = i * i + j * j;
    require(hsq >= ksq);
    const k = sqrt(ksq);
    require(is_integer(k));
    list(i, j, k);
}
```

4.3.2 Examples of Nondeterministic Programs

Section 4.3.3 describes the implementation of the amb evaluator. First, however, we give some examples of how it can be used. The advantage of nondeterministic programming is that we can suppress the details of how search is carried out, thereby expressing our programs at a higher level of abstraction.

Logic Puzzles

The following puzzle (taken from Dinesman 1968) is typical of a large class of simple logic puzzles:

> Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?

We can determine who lives on each floor in a straightforward way by enumerating all the possibilities and imposing the given restrictions:[39]

```
function multiple_dwelling() {
    const baker = amb(1, 2, 3, 4, 5);
    const cooper = amb(1, 2, 3, 4, 5);
    const fletcher = amb(1, 2, 3, 4, 5);
    const miller = amb(1, 2, 3, 4, 5);
    const smith = amb(1, 2, 3, 4, 5);
    require(distinct(list(baker, cooper, fletcher, miller, smith)));
    require(! baker === 5);
    require(! cooper === 1);
    require(! fletcher === 5);
    require(! fletcher === 1);
    require(! miller > cooper);
```

[39] Our program uses the following function to determine if the elements of a list are distinct:

```
function distinct(items) {
    return is_null items
        ? true
        : is_null(tail(items))
          ? true
          : is_null(member(head(items), tail(items)))
            ? distinct(tail(items))
            : false;
}
```

The function member_equal is like member except that it uses equal instead of === to test for equality.

```
    require(! math_abs(smith - fletcher) === 1);
    require(! math_abs(fletcher - cooper) === 1);
    return list(list("baker", baker),
                list("cooper", cooper),
                list("fletcher", fletcher),
                list("miller", miller),
                list("smith", smith));
}
```

Evaluating the expression `multiple_dwelling()` produces the result

```
[["baker", [3, null]],
 [["cooper", [2, null]],
  [["fletcher", [4, null]],
   [["miller", [5, null]],
    [["smith", [1, null]], null]]]]]
```

Although this simple function works, it is very slow. Exercises 4.30 and 4.31 discuss some possible improvements.

Exercise 4.29

Modify the multiple-dwelling function to omit the requirement that Smith and Fletcher do not live on adjacent floors. How many solutions are there to this modified puzzle?

Exercise 4.30

Does the order of the restrictions in the multiple-dwelling function affect the answer? Does it affect the time to find an answer? If you think it matters, demonstrate a faster program obtained from the given one by reordering the restrictions. If you think it does not matter, argue your case.

Exercise 4.31

In the multiple dwelling problem, how many sets of assignments are there of people to floors, both before and after the requirement that floor assignments be distinct? It is very inefficient to generate all possible assignments of people to floors and then leave it to backtracking to eliminate them. For example, most of the restrictions depend on only one or two of the person-floor variables, and can thus be imposed before floors have been selected for all the people. Write and demonstrate a much more efficient nondeterministic function that solves this problem based upon generating only those possibilities that are not already ruled out by previous restrictions.

Exercise 4.32

Write an ordinary JavaScript program to solve the multiple dwelling puzzle.

Exercise 4.33

Solve the following 'Liars' puzzle (from Phillips 1934):

> Five schoolgirls sat for an examination. Their parents—so they thought—showed an undue degree of interest in the result. They therefore agreed that, in writing home about the examination, each girl should make one true statement and one untrue one. The following are the relevant passages from their letters:
>
> - Betty: 'Kitty was second in the examination. I was only third.'
> - Ethel: 'You'll be glad to hear that I was on top. Joan was second.'
> - Joan: 'I was third, and poor old Ethel was bottom.'
> - Kitty: 'I came out second. Mary was only fourth.'
> - Mary: 'I was fourth. Top place was taken by Betty.'
>
> What in fact was the order in which the five girls were placed?

Exercise 4.34

Use the `amb` evaluator to solve the following puzzle:[40]

> Mary Ann Moore's father has a yacht and so has each of his four friends: Colonel Downing, Mr. Hall, Sir Barnacle Hood, and Dr. Parker. Each of the five also has one daughter and each has named his yacht after a daughter of one of the others. Sir Barnacle's yacht is the Gabrielle, Mr. Moore owns the Lorna; Mr. Hall the Rosalind. The Melissa, owned by Colonel Downing, is named after Sir Barnacle's daughter. Gabrielle's father owns the yacht that is named after Dr. Parker's daughter. Who is Lorna's father?

Try to write the program so that it runs efficiently (see exercise 4.31). Also determine how many solutions there are if we are not told that Mary Ann's last name is Moore.

[40] This is taken from a booklet called 'Problematical Recreations,' published in the 1960s by Litton Industries, where it is attributed to the *Kansas State Engineer*.

Exercise 4.35

Exercise 2.26 described the 'eight-queens puzzle' of placing queens on a chessboard so that no two attack each other. Write a nondeterministic program to solve this puzzle.

Parsing natural language

Programs designed to accept natural language as input usually start by attempting to *parse* the input, that is, to match the input against some grammatical structure. For example, we might try to recognize simple sentences consisting of an article followed by a noun followed by a verb, such as 'The cat eats.' To accomplish such an analysis, we must be able to identify the parts of speech of individual words. We could start with some lists that classify various words:[41]

```
const nouns = list("noun", "student", "professor", "cat", "class");

const verbs = list("verb", "studies", "lectures", "eats", "sleeps");

const articles = list("article", "the", "a");
```

We also need a *grammar*, that is, a set of rules describing how grammatical elements are composed from simpler elements. A very simple grammar might stipulate that a sentence always consists of two pieces—a noun phrase followed by a verb—and that a noun phrase consists of an article followed by a noun. With this grammar, the sentence 'The cat eats' is parsed as follows:

```
["sentence", [["noun-phrase", [["article", ["the", null]],
                               ["noun", ["cat", null]]]],
              ["verb", ["eats", null]]]]
```

We can generate such a parse with a simple program that has separate functions for each of the grammatical rules. To parse a sentence, we identify its two constituent pieces and return a list of these two elements, tagged with the symbol `sentence`:

```
function parse_sentence() {
    return list("sentence",
                parse_noun_phrase(),
                parse_word(verbs));
}
```

A noun phrase, similarly, is parsed by finding an article followed by a noun:

```
function parse_noun_phrase() {
    return list("noun-phrase",
```

[41] Here we use the convention that the first element of each list designates the part of speech for the rest of the words in the list.

```
                parse_word(articles),
                parse_word(nouns));
}
```

At the lowest level, parsing boils down to repeatedly checking that the next unparsed word is a member of the list of words for the required part of speech. To implement this, we maintain a global variable unparsed, which is the input that has not yet been parsed. Each time we check a word, we require that unparsed, must be non-empty and that it should begin with a word from the designated list. If so, we remove that word from unparsed, and return the word together with its part of speech (which is found at the head of the list):[42]

```
function parse_word(word_list) {
    require(! is_null(unparsed));
    require(member(head(unparsed), tail(word_list)) !== null);
    const found_word = head(unparsed);
    unparsed = tail(unparsed);
    return list(head(word_list), found_word);
}
```

To start the parsing, all we need to do is set unparsed, to be the entire input, try to parse a sentence, and check that nothing is left over:

```
let unparsed = null;

function parse_input(input) {
    unparsed = input;
    const sent = parse_sentence();
    require(is_null(unparsed));
    return sent;
}
```

We can now try the parser and verify that it works for our simple test sentence:

```
// Amb-Eval input:
parse_input(list("the",  "cat",  "eats"));
// Starting a new problem
// Amb-Eval value:
["sentence", [["noun-phrase", [["article", ["the", null]],
                               ["noun", ["cat", null]]]],
              ["verb", ["eats", null]]]]
```

The amb evaluator is useful here because it is convenient to express the parsing constraints with the aid of require. Automatic search and backtracking really pay off, however, when we consider more complex grammars where there are choices for how the units can be decomposed.

[42] Notice that parse_word, uses assignment to modify the unparsed input list. For this to work, our amb evaluator must undo the effects of assignment operations when it backtracks.

Let's add to our grammar a list of prepositions:

```
const prepositions = list("prep", "for", "to",  "in", "by", "with");
```

and define a prepositional phrase (e.g., 'for the cat') to be a preposition followed by a noun phrase:

```
function parse_prepositional_phrase() {
    return list("prep-phrase",
                parse_word(prepositions),
                parse_noun_phrase());
}
```

Now we can define a sentence to be a noun phrase followed by a verb phrase, where a verb phrase can be either a verb or a verb phrase extended by a prepositional phrase:[43]

```
function parse_sentence() {
    return list("sentence",


}

function parse_verb_phrase() {
    function maybe_extend(verb_phrase) {
        return amb(verb_phrase,
                   maybe_extend(list("verb-phrase",
                                     verb_phrase,
                                     parse_prepositional_phrase())));
    }
    return maybe_extend(parse_word(verbs));
}
```

While we're at it, we can also elaborate the definition of noun phrases to permit such things as 'a cat in the class.' What we used to call a noun phrase, we'll now call a simple noun phrase, and a noun phrase will now be either a simple noun phrase or a noun phrase extended by a prepositional phrase:

```
function parse_simple_noun_phrase() {
    return list("simple-noun-phrase",
                parse_word(articles),
                parse_word(nouns));
}

function parse_noun_phrase() {
    function maybe_extend(noun_phrase) {
        return amb(noun_phrase,
                   maybe_extend(list("noun-phrase",
                                     noun_phrase,
```

[43] Observe that this definition is recursive—a verb may be followed by any number of prepositional phrases.

```
                                        parse_prepositional_phrase()));
    }
    return maybe_extend(parse_simple_noun_phrase());
}
```

Our new grammar lets us parse more complex sentences. For example

```
parse_input(list("the", "student", "with", "the", "cat", "sleeps", "in", "the", "cla
```

produces

```
["sentence",
 [["noun-phrase",
  [["simple-noun-phrase",
    [["article", ["the", null]],
     [["noun", ["student", null]],
      null],
     ["prep-phrase",
      [["prep", ["with", null]],
       [["simple-noun-phrase",
        [["article", ["the", null]],
         [["noun", ["cat", null]]],
          null]]]]]],
     [["verb-phrase",
      [["verb", ["sleeps", null]],
       [["prep-phrase",
        [["prep", ["in", null]],
         [["simple-noun-phrase",
          [["article", ["the", null]],
           ["noun", ["class", null]]]]]]]]]]]]]
```

Observe that a given input may have more than one legal parse. In the sentence 'The professor lectures to the student with the cat,' it may be that the professor is lecturing with the cat, or that the student has the cat. Our nondeterministic program finds both possibilities:

```
parse_input(list("the", "professor", "lectures",
                 "to", "the", "student", "with", "the", "cat"));
```

produces

```
["sentence",
 [["simple-noun-phrase",
   [["article", ["the", null]],
    [["noun", ["professor", null]],
     null]],
    null]],
   [["verb-phrase",
    [["verb-phrase",
     [["verb", ["lectures", null]],
      [["prep-phrase",
```

```
        [["prep", ["to", null]],
         ["simple-noun-phrase",
          [["article", ["the", null]],
           [["noun", ["student", null]],
       [["prep-phrase",
         [["prep", ["with", null]],
          ["simple-noun-phrase",
           [["article", ["the", null]],
            [["noun", ["cat", null]]]]]]]]]]]
```

Asking the evaluator to try again yields

```
["sentence",
 [["simple-noun-phrase",
   [["article", ["the", null]],
    [["noun", ["professor", null]],
     null]],
    null]],
   [["verb-phrase",
    [["verb-phrase",
     [["verb", ["lectures", null]],
      [["prep-phrase",
        [["prep", ["to", null]],
         [["noun-phrase",
          [["simple-noun-phrase",
          [["article", ["the", null]],
           [["noun", ["student", null]],
          [["prep-phrase",
         [["prep", ["with", null]],
          ["simple-noun-phrase",
           [["article", ["the", null]],
            [["noun", ["cat", null]]]]]]]]]]]
```

Exercise 4.36

With the grammar given above, the following sentence can be parsed in five different ways: 'The professor lectures to the student in the class with the cat.' Give the five parses and explain the differences in shades of meaning among them.

Exercise 4.37

The evaluators in sections 4.1 and 4.2 do not determine what order operands are evaluated in. We will see that the `amb` evaluator evaluates them from left to right. Explain why our parsing program wouldn't work if the operands were evaluated in some other order.

Exercise 4.38

Louis Reasoner suggests that, since a verb phrase is either a verb or a verb phrase followed by a prepositional phrase, it would be much more straightforward to define the function `parse-verb-phrase` as follows (and similarly for noun phrases):

```
function parse_verb_phrase() {
    return amb(parse_word(verbs),
               list("verb-phrase",
                    parse_verb_phrase(),
                    parse_prepositional_phrase()));
}
```

Does this work? Does the program's behavior change if we interchange the order of expressions in the `amb`?

Exercise 4.39

Extend the grammar given above to handle more complex sentences. For example, you could extend noun phrases and verb phrases to include adjectives and adverbs, or you could handle compound sentences.[44]

Exercise 4.40

Alyssa P. Hacker is more interested in generating interesting sentences than in parsing them. She reasons that by simply changing the function `parse_word` so that it ignores the 'input sentence' and instead always succeeds and generates an appropriate word, we can use the programs we had built for parsing to do generation instead. Implement Alyssa's idea, and show the first half-dozen or so sentences generated.[45]

[44] This kind of grammar can become arbitrarily complex, but it is only a toy as far as real language understanding is concerned. Real natural-language understanding by computer requires an elaborate mixture of syntactic analysis and interpretation of meaning. On the other hand, even toy parsers can be useful in supporting flexible command languages for programs such as information-retrieval systems. Winston 1992 discusses computational approaches to real language understanding and also the applications of simple grammars to command languages.

[45] Although Alyssa's idea works just fine (and is surprisingly simple), the sentences that it generates are a bit boring—they don't sample the possible sentences of this language in a very interesting way. In fact, the grammar is highly recursive in many places, and Alyssa's technique 'falls into' one of these recursions and gets stuck. See exercise 4.41 for a way to deal with this.

4.3.3 Implementing the `amb` Evaluator

The evaluation of an ordinary JavaScript expression may return a value, may never terminate, or may signal an error. In nondeterministic JavaScript the evaluation of an expression may in addition result in the discovery of a dead end, in which case evaluation must backtrack to a previous choice point. The interpretation of nondeterministic JavaScript is complicated by this extra case.

We will construct the `amb` evaluator for nondeterministic JavaScript by modifying the analyzing evaluator of section 4.1.7.[46] As in the analyzing evaluator, evaluation of an expression is accomplished by calling an execution function produced by analysis of that expression. The difference between the interpretation of ordinary JavaScript and the interpretation of nondeterministic JavaScript will be entirely in the execution functions.

Execution functions and continuations

Recall that the execution functions for the ordinary evaluator take one argument: the environment of execution. In contrast, the execution functions in the `amb` evaluator take three arguments: the environment, and two functions called *continuation functions*. The evaluation of an expression will finish by calling one of these two continuations: If the evaluation results in a value, the *success continuation* is called with that value; if the evaluation results in the discovery of a dead end, the *failure continuation* is called. Constructing and calling appropriate continuations is the mechanism by which the nondeterministic evaluator implements backtracking.

It is the job of the success continuation to receive a value and proceed with the computation. Along with that value, the success continuation is passed another failure continuation, which is to be called subsequently if the use of that value leads to a dead end.

It is the job of the failure continuation to try another branch of the nondeterministic process. The essence of the nondeterministic language is in the fact that expressions may represent choices among alternatives. The evaluation of such an expression must proceed with one of the indicated alternative choices, even though it is not known in advance which choices will lead to acceptable results. To deal with this, the evaluator picks one of the alternatives and passes this value to the success continuation. Together with this value, the evaluator constructs and passes along a failure continuation that can be called later to choose a different alternative.

A failure is triggered during evaluation (that is, a failure continuation is called) when a user program explicitly rejects the current line of attack (for example, a call to `require` may result in

[46] We chose to implement the lazy evaluator in section 4.2 as a modification of the ordinary metacircular evaluator of section 4.1.1. In contrast, we will base the `amb` evaluator on the analyzing evaluator of section 4.1.7, because the execution functions in that evaluator provide a convenient framework for implementing backtracking.

execution of `amb()`, an expression that always fails—see section 4.3.1). The failure continuation in hand at that point will cause the most recent choice point to choose another alternative. If there are no more alternatives to be considered at that choice point, a failure at an earlier choice point is triggered, and so on. Failure continuations are also invoked by the driver loop in response to a **`try`**`-again` request, to find another value of the expression.

In addition, if a side-effect operation (such as assignment to a variable) occurs on a branch of the process resulting from a choice, it may be necessary, when the process finds a dead end, to undo the side effect before making a new choice. This is accomplished by having the side-effect operation produce a failure continuation that undoes the side effect and propagates the failure.

In summary, failure continuations are constructed by

- `amb` expressions—to provide a mechanism to make alternative choices if the current choice made by the `amb` expression leads to a dead end;
- the top-level driver—to provide a mechanism to report failure when the choices are exhausted;
- assignments—to intercept failures and undo assignments during backtracking.

Failures are initiated only when a dead end is encountered. This occurs

- if the user program executes `amb()`;
- if the user types **`try`**`-again` at the top-level driver.

Failure continuations are also called during processing of a failure:

- When the failure continuation created by an assignment finishes undoing a side effect, it calls the failure continuation it intercepted, in order to propagate the failure back to the choice point that led to this assignment or to the top level.
- When the failure continuation for an `amb` runs out of choices, it calls the failure continuation that was originally given to the `amb`, in order to propagate the failure back to the previous choice point or to the top level.

Structure of the evaluator

The syntax- and data-representation functions for the amb evaluator, and also the basic analyze function, are identical to those in the evaluator of section 4.1.7, except for the fact that we need additional syntax functions to recognize amb expressions:[47]

```
function is_amb(stmt) {
    return is_tagged_list(stmt, "application") &&
           is_name(operator(stmt)) &&
           name_of_name(operator(stmt)) === "amb";
}
function amb_choices(stmt) {
    return operands(stmt);
}
```

We must also add to the dispatch in analyze a clause that will recognize such expressions and generate an appropriate execution function:

```
: is_amb(stmt)
  ? analyze_amb(stmt)
```

The top-level function ambeval (similar to the version of evaluate given in section 4.1.7) analyzes the given expression and applies the resulting execution function to the given environment, together with two given continuations:

```
function ambeval(exp, env, succeed, fail) {
    return analyze(exp)(env, succeed, fail);
}
```

A success continuation is a function of two arguments: the value just obtained and another failure continuation to be used if that value leads to a subsequent failure. A failure continuation is a function of no arguments. So the general form of an execution function is

```
(env, succeed, fail) => {
    // succeed is a function of the form: (value, fail) => ...
    // fail    is a function of the form: is () => ...
}
```

For example, executing

```
ambeval(exp,
    the_global_environment,
    (value, fail) => value,
    () => "failed");
```

will attempt to evaluate the given expression and will return either the expression's value (if the evaluation succeeds) or the string "failed" (if the evaluation fails). The call to ambeval

[47] We assume that the evaluator supports **let** (see exercise **??**), which we have used in our nondeterministic programs.

in the driver loop shown below uses much more complicated continuation functions, which continue the loop and support the **`try`**`-again` request.

Most of the complexity of the `amb` evaluator results from the mechanics of passing the continuations around as the execution functions call each other. In going through the following code, you should compare each of the execution functions with the corresponding function for the ordinary evaluator given in section 4.1.7.

Simple expressions

The execution functions for the simplest kinds of expressions are essentially the same as those for the ordinary evaluator, except for the need to manage the continuations. The execution functions simply succeed with the value of the expression, passing along the failure continuation that was passed to them.

```
function analyze_self_evaluating(stmt) {
    return (env, succeed, fail) => succeed(stmt, fail);
}

function analyze_name(stmt) {
    return (env, succeed, fail) =>
           succeed(lookup_name_value(name_of_name(stmt), env),
                   fail);
}

function analyze_function_definition(stmt) {
    const vars =
        function_definition_parameters(stmt);
    const bfun =
        analyze(function_definition_body(stmt);
    return (env, succeed, fail) =>
        succeed(make_function(vars, bfun, env),
                fail);
}
```

Notice that looking up a name always ‘succeeds.’ If `lookup_name_value` fails to find the name, it signals an error, as usual. Such a ‘failure’ indicates a program bug—a reference to an unbound variable; it is not an indication that we should try another nondeterministic choice instead of the one that is currently being tried.

Conditionals and sequences

Conditionals are also handled in a similar way as in the ordinary evaluator. The execution function generated by `analyze_conditional_expression` invokes the predicate execution function `pfun` with a success continuation that checks whether the predicate value is true and goes on to execute either the consequent or the alternative. If the execution of `pfun` fails, the original failure continuation for the conditional expression is called.

```
function analyze_conditional_expression(stmt) {
    const pfun = analyze(cond_expr_pred(stmt);
    const cfun = analyze(cond_expr_cons(stmt));
    const afun = analyze(cond_expr_alt(stmt));
    return (env, succeed, fail) =>
           pfun(env,
                // success continuation for evaluating the predicate
                // to obtain pred_value
                (pred_value, fail2) =>
                    is_true(pred_value)
                    ? cfun(env, succeed, fail2)
                    : afun(env, succeed, fail2),
                fail);
}
```

Sequences are also handled in the same way as in the previous evaluator, except for the machinations in the subfunction `sequentially` that are required for passing the continuations. Namely, to sequentially execute `a` and then `b`, we call `a` with a success continuation that calls `b`.

```
function analyze_sequence(stmts) {
    function sequentially(a, b) {
        return (env, succeed, fail) =>
                   a(env,
                     (a_value, fail2) => b(env, succeed, fail2),
                     fail);
    }
    function loop(first_fun, rest_funs) {
        return is_null(rest_funs)
               ? first_fun
               : loop(sequentially(first_fun,
                                   head(rest_funs)),
                      tail(rest_funs));
    }
    const funs = map(analyze, stmts);
    return is_null(funs)
           ? env => undefined
           : loop(head(funs), tail(funs));
}
```

Declarations and assignments

Declarations are another case where we must go to some trouble to manage the continuations, because it is necessary to evaluate the definition-value expression before actually defining the new variable. To accomplish this, the declaration-value execution function `vfun` is called with the environment, a success continuation, and the failure continuation. If the execution of `vfun` succeeds, obtaining a value `val` for the declared name, the name is declared and the success is propagated:

```
function analyze_variable_declaration(stmt) {
    const name = variable_declaration_name(stmt);
    const vfun = variable_declaration_value(stmt);
    return (env, succeed, fail) =>
              vfun(env,
                   (val, fail2) => {
                       declare_variable(name, val, env);
                       succeed("ok", fail2);
                   },
                   fail);
}
function analyze_constant_declaration(stmt) {
    const name =
        constant_declaration_name(stmt);
    const vfun =
        constant_declaration_value(stmt);
    return (env, succeed, fail) =>
              vfun(env,
                   (val, fail2) => {
                       declare_constant(name, val, env);
                       succeed("ok", fail2);
                   },
                   fail);
}
```

Assignments are more interesting. This is the first place where we really use the continuations, rather than just passing them around. The execution function for assignments starts out like the one for definitions. It first attempts to obtain the new value to be assigned to the variable. If this evaluation of `vfun` fails, the assignment fails.

If `vfun` succeeds, however, and we go on to make the assignment, we must consider the possibility that this branch of the computation might later fail, which will require us to backtrack out of the assignment. Thus, we must arrange to undo the assignment as part of the backtracking process.[48]

[48] We didn't worry about undoing declarations, since we can assume that internal declarations are scanned out (section 4.1.6).

This is accomplished by giving `vfun` a success continuation (marked with the comment '*1*' below) that saves the old value of the variable before assigning the new value to the variable and proceeding from the assignment. The failure continuation that is passed along with the value of the assignment (marked with the comment '*2*' below) restores the old value of the variable before continuing the failure. That is, a successful assignment provides a failure continuation that will intercept a subsequent failure; whatever failure would otherwise have called `fail2` calls this function instead, to undo the assignment before actually calling `fail2`.

```
function analyze_assignment(stmt) {
    const variable = assignment_variable(stmt);
    const vfun = analyze(assignment_value(stmt));
    return (env, succeed, fail) =>
           vfun(env,
                (val, fail2) => { // *1*
                    const old_value = lookup_name_value(variable, env);
                    set_variable_value(variable, val, env);
                    succeed("ok",
                            () => { // *2*
                                set_variable_value(variable, old_value, env);
                                fail2();
                            });
                },
                fail);
}
```

Function applications

The execution function for applications contains no new ideas except for the technical complexity of managing the continuations. This complexity arises in `analyze_application`, due to the need to keep track of the success and failure continuations as we evaluate the operands. We use a function `get_args` to evaluate the list of operands, rather than a simple `map` as in the ordinary evaluator.

```
function analyze_application(stmt) {
    const ffun = analyze(operator(stmt));
    const afuns = map(analyze, operands(stmt));
    return (env, succeed, fail) =>
              ffun(env,
                   (fun, fail2) =>
                       get_args(afuns,
                                env,
                                (args, fail3) =>
                                    execute_application(fun,
                                        args, succeed, fail3),
                                fail2),
                   fail);
```

```
}
```

In get_args, notice how tailing down the list of afun, execution functions and pairing up the resulting list of args is accomplished by calling each afun in the list with a success continuation that recursively calls get_args. Each of these recursive calls to get_args has a success continuation whose value is the pair of the newly obtained argument onto the list of accumulated arguments:

```
function get_args(afuns, env, succeed, fail) {
    return is_null(afuns)
        ? succeed(null, fail)
        : head(afuns)(env,
                      // success continuation for this afun
                      (arg, fail2) =>
                          get_args(tail(afuns),
                                   env,
                                   // success continuation for
                                   // recursive call to get_args
                                   (args, fail3) =>
                                       succeed(pair(arg, args),
                                               fail3),
                                   fail2),
                      fail);
}
```

The actual function application, which is performed by execute_application, is accomplished in the same way as for the ordinary evaluator, except for the need to manage the continuations.

```
function execute_application(fun, args, succeed, fail) {
    return is_primitive_function(fun)
        ? succeed(apply_primitive_function(fun, args),
                  fail)
        : is_compound_function(fun)
        ? function_body(fun)(
              extend_environment(
                  function_parameters(fun),
                  args,
                  function_environment(fun)),
              succeed,
              fail)
        : error(fun, "unknown function type in " +
                     "execute_application");
}
```

Evaluating amb expressions

The amb expression is the key element in the nondeterministic language. Here we see the essence of the interpretation process and the reason for keeping track of the continuations. The execution function for amb defines a loop try_next that cycles through the execution functions for all the possible values of the amb expression. Each execution function is called with a failure continuation that will try the next one. When there are no more alternatives to try, the entire amb expression fails.

```
function analyze_amb(exp) {
    const cfuns = map(analyze, amb_choices(exp));
    return (env, succeed, fail) => {
        function try_next(choices) {
            return is_null(choices)
                ? fail()
                : head(choices)(env,
                                succeed,
                                () =>
                                try_next(tail(choices)));
        }
        return try_next(cfuns);
    };
}
```

Driver loop

The driver loop for the amb evaluator is complex, due to the mechanism that permits the user to try again in evaluating an expression. The driver uses a function called internal_loop, which takes as argument a function try_again. The intent is that calling try_again should go on to the next untried alternative in the nondeterministic evaluation. The function internal_loop either calls try_again in response to the user typing **try**-again at the driver loop, or else starts a new evaluation by calling ambeval.

The failure continuation for this call to ambeval informs the user that there are no more values and re-invokes the driver loop.

The success continuation for the call to ambeval is more subtle. We print the obtained value and then invoke the internal loop again with a try_again function that will be able to try the next alternative. This next_alternative function is the second argument that was passed to the success continuation. Ordinarily, we think of this second argument as a failure continuation to be used if the current evaluation branch later fails. In this case, however, we have completed a successful evaluation, so we can invoke the 'failure' alternative branch in order to search for additional successful evaluations.

```
const input_prompt = "// Amb-Eval input:");
```

```
const output_prompt =  "// Amb-Eval value:");
function driver_loop() {
    function internal_loop(try_again) {
        prompt_for_input(input_prompt);
        const input = read();
        if (input === "try-again") {
            try_again();
        } else {
            display("// Starting a new problem ");
            ambeval(input,
                the_global_environment,
                // ambeval success
                (val, next_alternative) => {
                    announce_output(output_prompt);
                    user_print(val);
                    return internal_loop(next_alternative);
                },
                // ambeval failure
                () => {
                    announce_output(
                        "// There are no more values of");
                    user_print(input);
                    return driver_loop();
                });
        }
    }
    return internal_loop(
               () => {
                   display("// There is no current problem");
                   return driver_loop();
               });
}
```

The initial call to `internal_loop` uses a `try_again` function that complains that there is no current problem and restarts the driver loop. This is the behavior that will happen if the user types `try-again` when there is no evaluation in progress.

Exercise 4.41

Implement a new special form `ramb` that is like `amb` except that it searches alternatives in a random order, rather than from left to right. Show how this can help with Alyssa's problem in exercise 4.40.

Exercise 4.42

Implement a new kind of assignment that is not undone upon failure. For example, we can

choose two distinct elements from a list and count the number of trials required to make a successful choice as follows:

```
let count = 0;

let x = an_element_of("a", "b", "c");
let y = an_element_of("a", "b", "c");
count = count + 1;
require(! x === y);
list(x, y, count);

// Starting a new problem
// Amb-Eval value:
["a", ["b", [2, null]]]

// Amb-Eval input:
try-again
// Amb-Eval value:
["a", ["c", [3, null]]]
```

What values would have been displayed if we had used assignment as originally defined here rather than the new kind?

Exercise 4.43

Implement a new kind of expression `if_fail` that permits the user to catch the failure of an expression. An application of `if_fail` to two expressions evaluates the first one as usual, and returns as usual if the evaluation succeeds. If the evaluation fails, however, the value of the second expression is returned, as in the following example:

```
// Amb-Eval input:
if_fail( ( x => {
               require(is_even(x));
               return x;
           } )(an_element_of(1, 3, 5)),
         "all-odd");
// Starting a new problem
// Amb-Eval value:
"all-odd"
// Amb-Eval input:
if_fail( ( x => {
               require(is_even(x));
               return x;
           } )(an_element_of(1, 3, 5, 8)),
         "all-odd");
// Starting a new problem
// Amb-Eval value:
8
```

Exercise 4.44

With the new kind of assignment as described in exercise 4.42 and `if_fail` as in exercise 4.43, what will be the result of evaluating

```
let pairs = null;
if_fail( ( p => {
               pairs = pair(p, pairs);
               amb();
           } )(prime_sum_pair(list(1, 3, 5, 8), list(20, 35, 110))),
         pairs);
```

Exercise 4.45

If we had not realized that `require` could be implemented as an ordinary function that uses `amb`, to be defined by the user as part of a nondeterministic program, we would have had to implement it in a way similar to the way we implemented `amb`. This would require syntax functions

```
function is_require(stmt) {
    return is_tagged_list(stmt, "application") &&
           is_name(operator(stmt)) &&
           name_of_name(operator(stmt)) === "require";
}
function require_predicate(exp) {
    return operands(exp);
}
```

and a new clause in the dispatch in `analyze`

```
: is_require(stmt)
  ? analyze_require(stmt)
```

as well the function `analyze-require` that handles `require` expressions. Complete the following definition of `analyze-require`.

```
function analyze_require(stmt) {
    return (env, succeed, fail) =>
        pfun(env,
             (pred_value, fail2) =>
                 ...
                 ? ...
                 : succeed("ok", fail2),
             fail);
}
```

4.4 Logic Programming

Note: this section is a work in progress!

In chapter 1 we stressed that computer science deals with imperative (how to) knowledge, whereas mathematics deals with declarative (what is) knowledge. Indeed, programming languages require that the programmer express knowledge in a form that indicates the step-by-step methods for solving particular problems. On the other hand, high-level languages provide, as part of the language implementation, a substantial amount of methodological knowledge that frees the user from concern with numerous details of how a specified computation will progress.

Most programming languages, including are organized around computing the values of mathematical functions. Expression-oriented languages (such as Lisp, Fortran, and Algol) capitalize on the 'pun' that an expression that describes the value of a function may also be interpreted as a means of computing that value. Because of this, most programming languages are strongly biased toward unidirectional computations (computations with well-defined inputs and outputs). There are, however, radically different programming languages that relax this bias. We saw one such example in section 3.3.5, where the objects of computation were arithmetic constraints. In a constraint system the direction and the order of computation are not so well specified; in carrying out a computation the system must therefore provide more detailed 'how to' knowledge than would be the case with an ordinary arithmetic computation. This does not mean, however, that the user is released altogether from the responsibility of providing imperative knowledge. There are many constraint networks that implement the same set of constraints, and the user must choose from the set of mathematically equivalent networks a suitable network to specify a particular computation.

The nondeterministic program evaluator of section 4.3 also moves away from the view that programming is about constructing algorithms for computing unidirectional functions. In a nondeterministic language, expressions can have more than one value, and, as a result, the computation is dealing with relations rather than with single-valued functions. Logic programming extends this idea by combining a relational vision of programming with a powerful kind of symbolic pattern matching called *unification*.[49]

[49]Logic programming has grown out of a long history of research in automatic theorem proving. Early theorem-proving programs could accomplish very little, because they exhaustively searched the space of possible proofs. The major breakthrough that made such a search plausible was the discovery in the early 1960s of the *unification algorithm* and the *resolution principle* (Robinson 1965). Resolution was used, for example, by Green and Raphael (1968) (see also Green 1969) as the basis for a deductive question-answering system. During most of this period, researchers concentrated on algorithms that are guaranteed to find a proof if one exists. Such algorithms were difficult to control and to direct toward a proof. Hewitt (1969) recognized the possibility of merging the control structure of a programming language with the operations of a logic-manipulation system, leading to the work in automatic search mentioned in section 4.3.1 (footnote 38). At the same time that this was being done,

This approach, when it works, can be a very powerful way to write programs. Part of the power comes from the fact that a single 'what is' fact can be used to solve a number of different problems that would have different 'how to' components. As an example, consider the append operation, which takes two lists as arguments and combines their elements to form a single list. In a procedural language such as JavaScript, we could define append in terms of the basic list constructor pair, as we did in section 2.1.1:

```
function append(x, y) {
    return is_null(x)
        ? y
        : pair(head(x), append(tail(x), y));
}
```

This function can be regarded as a translation into JavaScript of the following two rules, the first of which covers the case where the first list is empty and the second of which handles the case of a nonempty list, which is a pair of two parts:

- For any list y, the empty list and y append to form y.
- For any u, v, y, and z, pair(u, v) and y append to form pair(u, z) if v and y append to form z.[50]

Using the append function, we can answer questions such as

> Find the append of list("a", "b") and list("c", "d").

But the same two rules are also sufficient for answering the following sorts of questions, which the function can't answer:

> Find a list y that appends with list("a", "b") to produce list("a", "b", "c", "d").
> Find all x and y that append to form list("a", "b", "c", "d").

In a logic programming language, the programmer writes an append 'function' by stating the two rules about append given above. 'How to' knowledge is provided automatically by the

Colmerauer, in Marseille, was developing rule-based systems for manipulating natural language (see Colmerauer et al. 1973). He invented a programming language called Prolog for representing those rules. Kowalski (1973; 1979), in Edinburgh, recognized that execution of a Prolog program could be interpreted as proving theorems (using a proof technique called linear Horn-clause resolution). The merging of the last two strands led to the logic-programming movement. Thus, in assigning credit for the development of logic programming, the French can point to Prolog's genesis at the University of Marseille, while the British can highlight the work at the University of Edinburgh. According to people at MIT, logic programming was developed by these groups in an attempt to figure out what Hewitt was talking about in his brilliant but impenetrable Ph.D. thesis. For a history of logic programming, see Robinson 1983.

[50] To see the correspondence between the rules and the function, let x in the function (where x is nonempty) correspond to pair(u, v) in the rule. Then z in the rule corresponds to the append of tail(x) and y.

interpreter to allow this single pair of rules to be used to answer all three types of questions about append.[51]

Contemporary logic programming languages (including the one we implement here) have substantial deficiencies, in that their general 'how to' methods can lead them into spurious infinite loops or other undesirable behavior. Logic programming is an active field of research in computer science.[52]

Earlier in this chapter we explored the technology of implementing interpreters and described the elements that are essential to an interpreter for a JavaScript-like language (indeed, to an interpreter for any conventional language). Now we will apply these ideas to discuss an interpreter for a logic programming language. We call this language the *query language*, because it is very useful for retrieving information from data bases by formulating *queries*, or questions, expressed in the language. Even though the query language is very different from JavaScript, we will find it convenient to describe the language in terms of the same general framework we have been using all along: as a collection of primitive elements, together with means of combination that enable us to combine simple elements to create more complex elements and means of abstraction that enable us to regard complex elements as single conceptual units. An interpreter for a logic programming language is considerably more complex than an interpreter for a language like JavaScript. Nevertheless, we will see that our query-language interpreter contains many of the same elements found in the interpreter of section 4.1. In particular, there will be an 'eval' part that classifies expressions according to type and an 'apply' part that implements the language's abstraction mechanism (functions in the case of JavaScript, and *rules* in the case of logic programming). Also, a central role is played in the implementation by a frame data structure, which determines the correspondence between symbols and their associated values. One additional interesting aspect of our query-language implementation is that we make substantial use of streams, which were introduced in chapter 3.

[51] This certainly does not relieve the user of the entire problem of how to compute the answer. There are many different mathematically equivalent sets of rules for formulating the append relation, only some of which can be turned into effective devices for computing in any direction. In addition, sometimes 'what is' information gives no clue 'how to' compute an answer. For example, consider the problem of computing the y such that $y^2 = x$.

[52] Interest in logic programming peaked during the early 80s when the Japanese government began an ambitious project aimed at building superfast computers optimized to run logic programming languages. The speed of such computers was to be measured in LIPS (Logical Inferences Per Second) rather than the usual FLOPS (FLoating-point Operations Per Second). Although the project succeeded in developing hardware and software as originally planned, the international computer industry moved in a different direction. See Feigenbaum and Shrobe 1993 for an overview evaluation of the Japanese project. The logic programming community has also moved on to consider relational programming based on techniques other than simple pattern matching, such as the ability to deal with numerical constraints such as the ones illustrated in the constraint-propagation system of section 3.3.5.

4.4.1 Deductive Information Retrieval

Logic programming excels in providing interfaces to data bases for information retrieval. The query language we shall implement in this chapter is designed to be used in this way.

In order to illustrate what the query system does, we will show how it can be used to manage the data base of personnel records for Microshaft, a thriving high-technology company in the Boston area. The language provides pattern-directed access to personnel information and can also take advantage of general rules in order to make logical deductions.

A sample data base

The personnel data base for Microshaft contains *assertions* about company personnel. Here is the information about Ben Bitdiddle, the resident computer wizard:

```
address(list("Bitdiddle", "Ben"), list("Slumerville", "Ridge Road", 10));
job(list("Bitdiddle", "Ben"), list("computer", "wizard"));
salary(list("Bitdiddle", "Ben"), 60000);
```

Each assertion is an application whose arguments can be expressions such as lists.

As resident wizard, Ben is in charge of the company's computer division, and he supervises two programmers and one technician. Here is the information about them:

```
address(list("Hacker", "Alyssa", "P"), list("Cambridge", "Mass Ave", 78));
job(list("Hacker", "Alyssa", "P"), list("computer", "programmer"));
salary(list("Hacker", "Alyssa", "P"), 40000);
supervisor(list("Hacker", "Alyssa", "P"), list("Bitdiddle", "Ben"));

address(list("Fect", "Cy", "D"), list("Cambridge", "Ames Street", 3));
job(list("Fect", "Cy", "D"), list("computer", "programmer"));
salary(list("Fect", "Cy", "D"), 35000);
supervisor(list("Fect", "Cy", "D"), list("Bitdiddle", "Ben"));

address(list("Tweakit", "Lem", "E"), list("Boston", "Bay State Road", 22));
job(list("Tweakit", "Lem", "E"), list("computer", "technician"));
salary(list("Tweakit", "Lem", "E"), 25000);
supervisor(list("Tweakit", "Lem", "E"), list("Bitdiddle", "Ben"));
```

There is also a programmer trainee, who is supervised by Alyssa:

```
address(list("Reasoner", "Louis"), list("Slumerville", "Pine Tree Road", 80));
job(list("Reasoner", "Louis"), list("computer", "programmer trainee"));
salary(list("Reasoner", "Louis"), 30000);
supervisor(list("Reasoner", "Louis"), list("Hacker", "Alyssa", "P"));
```

All of these people are in the computer division, as indicated by the word `computer` as the first item in their job descriptions. Ben is a high-level employee. His supervisor is the company's

big wheel himself:

```
supervisor(list("Bitdiddle", "Ben"), list("Warbucks", "Oliver"));

address(list("Warbucks", "Oliver"), list("Swellesley", "Top Heap Road"));
job(list("Warbucks", "Oliver"), list("administration", "big", "wheel"));
salary(list("Warbucks", "Oliver"), 150000);
```

Besides the computer division supervised by Ben, the company has an accounting division, consisting of a chief accountant and his assistant:

```
address(list("Scrooge", "Eben"), list("Weston", "Shady Lane", 10));
job(list("Scrooge", "Eben"), list("accounting", list("chief", "accountant")));
salary(list("Scrooge", "Eben"), 75000);
supervisor(list("Scrooge", "Eben"), list("Warbucks", "Oliver"));

address(list("Cratchet", "Robert"), list("Allston", "N Harvard Street", 16));
job(list("Cratchet", "Robert"), list("accounting", "scrivener"));
salary(list("Cratchet", "Robert"), 18000);
supervisor(list("Cratchet", "Robert"), list("Scrooge", "Eben"));
```

There is also a secretary for the big wheel:

```
address(list("Aull", "DeWitt"), list("Slumerville", "Onion Square", 5));
job(list("Aull", "DeWitt"), list("administration", "secretary"));
salary(list("Aull", "DeWitt"), 25000);
supervisor(list("Aull", "DeWitt"), list("Warbucks", "Oliver"));
```

The data base also contains assertions about which kinds of jobs can be done by people holding other kinds of jobs. For instance, a computer wizard can do the jobs of both a computer programmer and a computer technician:

```
can_do_job(list("computer", "wizard"), list("computer", "programmer"));
can_do_job(list("computer", "wizard"), list("computer", "technician"));
```

A computer programmer could fill in for a trainee:

```
can_do_job(list("computer", "programmer"),
           list("computer", "programmer", "trainee"));
```

Also, as is well known,

```
can_do_job(list("administration", "secretary"),
           list("administration", "big", "wheel"));
```

Simple queries

The query language allows users to retrieve information from the data base by posing queries in response to the system's prompt. For example, to find all computer programmers one can say

```
// Query input:
job(x, list("computer", "programmer"));
```

The system will respond with the following items:

```
// Query results:
job(list("Hacker", "Alyssa", "P"), list("computer", "programmer"))
job(list("Fect", "Cy", "D"), list("computer", "programmer"))
```

The input query specifies that we are looking for entries in the data base that match a certain *pattern*. In this example, the pattern specifies entries consisting of three items, of which the first is the literal symbol `job`, the second can be anything, and the third is the literal list `list("computer", "programmer")`. The 'anything' that can be the second item in the matching list is specified by a *pattern variable*, `x`. The system responds to a simple query by showing all entries in the data base that match the specified pattern.

A pattern can have more than one variable. For example, the query

```
address(x, y)
```

will list all the employees' addresses.

A pattern can have no variables, in which case the query simply determines whether that pattern is an entry in the data base. If so, there will be one match; if not, there will be no matches.

The same pattern variable can appear more than once in a query, specifying that the same 'anything' must appear in each position. This is why variables have names. For example,

```
supervisor(x, x)
```

finds all people who supervise themselves (though there are no such assertions in our sample data base).

The query

```
job(x, list("computer",  type));
```

matches all job entries whose third item is a two-element list whose first item is `"computer"`:

```
job(list("Bitdiddle", "Ben"), list("computer", "wizard"));
job(list("Hacker", "Alyssa", "P"), list("computer", "programmer"));
job(list("Fect", "Cy", "D"), list("computer", "programmer"));
job(list("Tweakit", "Lem", "E"), list("computer", "technician"));
```

This same pattern does *not* match

```
job(list("Reasoner", "Louis"), list("computer", "programmer", "trainee"));
```

because the third item in the entry is a list of three elements, and the pattern's third item specifies that there should be two elements. If we wanted to change the pattern so that the third item could be any list beginning with `computer`, we could specify

```
job(x, pair("computer", type))
```

For example,

```
pair("computer", type)
```

matches the data

```
list("computer", "programmer", "trainee")
```

with type as the list `list("programmer", "trainee")`. It also matches the data

```
list("computer", "programmer")
```

with type as the list `list("programmer")`, and matches the data

```
list("computer")
```

with type as the empty list `null`.

We can describe the query language's processing of simple queries as follows:

- The system finds all assignments to variables in the query pattern that *satisfy* the pattern—that is, all sets of values for the variables such that if the pattern variables are *instantiated with* (replaced by) the values, the result is in the data base.
- The system responds to the query by listing all instantiations of the query pattern with the variable assignments that satisfy it.

Note that if the pattern has no variables, the query reduces to a determination of whether that pattern is in the data base. If so, the empty assignment, which assigns no values to variables, satisfies that pattern for that data base.

Exercise 4.46

Give simple queries that retrieve the following information from the data base:

a. all people supervised by Ben Bitdiddle;

b. the names and jobs of all people in the accounting division;

c. the names and addresses of all people who live in Slumerville.

Compound queries

Simple queries form the primitive operations of the query language. In order to form compound operations, the query language provides means of combination. One thing that makes the query language a logic programming language is that the means of combination mirror the means of combination used in forming logical expressions: `and`, `or`, and `not`. (Here `and`, `or`, and `not` are not the JavaScript primitives, but rather operations built into the query language.)

We can use `and` as follows to find the addresses of all the computer programmers:

```
and(job(person, list("computer", "programmer"),
    address(person, where));
```

The resulting output is

```
and(job(list("Hacker", "Alyssa", "P"), list("computer", "programmer")),
    address(list("Hacker", "Alyssa", "P"), list("Cambridge", "Mass Ave", 78)))

and(job(list("Fect", "Cy", "D"), list("computer", "programmer")),
    address(list("Fect", "Cy", "D"), list("Cambridge", "Ames Street", 3)))
```

In general,

`and(`$\texttt{query}_1$`, `$\texttt{query}_2$`, ..., `$\texttt{query}_n$`);`

is satisfied by all sets of values for the pattern variables that simultaneously satisfy $query_1 \ldots query_n$.

As for simple queries, the system processes a compound query by finding all assignments to the pattern variables that satisfy the query, then displaying instantiations of the query with those values.

Another means of constructing compound queries is through `or`. For example,

```
or(supervisor(x, list("Bitdiddle", "Ben")),
   supervisor(x, list("Hacker", "Alyssa", "P")));
```

will find all employees supervised by Ben Bitdiddle or Alyssa P. Hacker:

```
or(supervisor(list("Hacker", "Alyssa", "P"), list("Bitdiddle", "Ben")),
supervisor(list("Hacker", "Alyssa", "P"), list("Hacker", "Alyssa", "P")))

or(supervisor(list("Fect", "Cy", "D"), list("Bitdiddle", "Ben")),
   supervisor(list("Fect", "Cy", "D"), list("Hacker", "Alyssa", "P")))

or(supervisor(list("Tweakit", "Lem", "E"), list("Bitdiddle", "Ben")),
   supervisor(list("Tweakit", "Lem", "E"), list("Hacker", "Alyssa", "P")))

or(supervisor(list("Reasoner", "Louis"), list("Bitdiddle", "Ben")),
   supervisor(list("Reasoner", "Louis"), list("Hacker", "Alyssa", "P")))
```

In general,

```
or(query_1, query_2, ..., query_n^);
```

is satisfied by all sets of values for the pattern variables that satisfy at least one of $query_1 \ldots query_n$.

Compound queries can also be formed with `not`. For example,

```
and(supervisor(x, list("Bitdiddle", "Ben")),
    not(job(x, list("computer", "programmer"))));
```

finds all people supervised by Ben Bitdiddle who are not computer programmers. In general,

```
not(query_1);
```

is satisfied by all assignments to the pattern variables that do not satisfy $query_1$.[53]

The final combining form is called `javascript_value`. When `javascript_value` is the first element of a pattern, it specifies that the next element is a JavaScript predicate to be applied to the rest of the (instantiated) elements as arguments. In general,

```
javascript_value(predicate, arg_1, ..., arg_n)
```

will be satisfied by assignments to the pattern variables for which the *predicate* applied to the instantiated $arg_1, \ldots, arg_n$ is true. For example, to find all people whose salary is greater than $30,000 we could write[54]

```
and(salary(person, amount),
    javascript_value(greater_than, amount, 30000));
```

Exercise 4.47

Formulate compound queries that retrieve the following information:

a. the names of all people who are supervised by Ben Bitdiddle, together with their addresses;

b. all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben Bitdiddle's salary;

c. all people who are supervised by someone who is not in the computer division, together with the supervisor's name and job.

[53] Actually, this description of `not`. is valid only for simple cases. The real behavior of `not` is more complex. We will examine `not`'s peculiarities in sections 4.4.2 and 4.4.3.

[54] Such `javascript_value` expressions should be used only to perform an operation not provided in the query language. In particular, it should not be used to test equality (since that is what the matching in the query language is designed to do) or inequality (since that can be done with the `same` rule shown below).

Rules

In addition to primitive queries and compound queries, the query language provides means for abstracting queries. These are given by *rules*. The rule

```
rule(lives_near(person_1, person_2),
     and(address(person_1, pair(town, rest_1)),
         address(person_2, pair(town, rest_2)),
         not(same(person_1, person_2))));
```

specifies that two people live near each other if they live in the same town. The final `not` clause prevents the rule from saying that all people live near themselves. The `same` relation is defined by a very simple rule:[55]

```
rule(same(x, x));
```

The following rule declares that a person is a 'wheel' in an organization if he supervises someone who is in turn a supervisor:

```
rule(wheel(person),
     and(supervisor(middle_manager, person),
         supervisor(x, middle_manager)));
```

The general form of a rule is

```
rule(conclusion, body);
```

where *conclusion* is a pattern and *body* is any query.[56] We can think of a rule as representing a large (even infinite) set of assertions, namely all instantiations of the rule conclusion with variable assignments that satisfy the rule body. When we described simple queries (patterns), we said that an assignment to variables satisfies a pattern if the instantiated pattern is in the data base. But the pattern needn't be explicitly in the data base as an assertion. It can be an implicit assertion implied by a rule. For example, the query

```
lives_near(x, list("Bitdiddle", "Ben"))
```

results in

```
lives_near(list("Reasoner", "Louis"), list("Bitdiddle", "Ben"))
lives_near(list("Aull", "DeWitt"), list("Bitdiddle", "Ben"))
```

To find all computer programmers who live near Ben Bitdiddle, we can ask

[55] Notice that we do not need `same` in order to make two things be the same: We just use the same pattern variable for each—in effect, we have one thing instead of two things in the first place. For example, see in the rule and in the `wheel` rule below. The `same` relation is useful when we want to force two things to be different, such as and in the rule. Although using the same pattern variable in two parts of a query forces the same value to appear in both places, using different pattern variables does not force different values to appear. (The values assigned to different pattern variables may be the same or different.)

[56] We will also allow rules without bodies, as in `same`, and we will interpret such a rule to mean that the rule conclusion is satisfied by any values of the variables.

```
and(job(x, list("computer", "programmer")),
    lives_near(x, list("Bitdiddle", "Ben")));
```

As in the case of compound functions, rules can be used as parts of other rules (as we saw with the `lives-near` rule above) or even be defined recursively. For instance, the rule

```
rule(outranked_by(staff_person, boss),
     or(supervisor(staff_person, boss),
        and(supervisor(staff_person, middle_manager),
            outranked_by(middle_manager, boss))));
```

says that a staff person is outranked by a boss in the organization if the boss is the person's supervisor or (recursively) if the person's supervisor is outranked by the boss.

Exercise 4.48

Define a rule that says that person 1 can replace person 2 if either person 1 does the same job as person 2 or someone who does person 1's job can also do person 2's job, and if person 1 and person 2 are not the same person. Using your rule, give queries that find the following:

a. all people who can replace Cy D. Fect;

b. all people who can replace someone who is being paid more than they are, together with the two salaries.

Exercise 4.49

Define a rule that says that a person is a 'big shot' in a division if the person works in the division but does not have a supervisor who works in the division.

Exercise 4.50

Ben Bitdiddle has missed one meeting too many. Fearing that his habit of forgetting meetings could cost him his job, Ben decides to do something about it. He adds all the weekly meetings of the firm to the Microshaft data base by asserting the following:

```
meeting("accounting", list("Monday", "9am"))
meeting("administration", list("Monday", "10am"))
meeting("computer", list("Wednesday", "3pm"))
meeting("administration", list("Friday", "1pm"))
```

Each of the above assertions is for a meeting of an entire division. Ben also adds an entry for the company-wide meeting that spans all the divisions. All of the company's employees attend this meeting.

```
meeting("whole-company", list("Wednesday", "4pm"))
```

a. On Friday morning, Ben wants to query the data base for all the meetings that occur that day. What query should he use?

b. Alyssa P. Hacker is unimpressed. She thinks it would be much more useful to be able to ask for her meetings by specifying her name. So she designs a rule that says that a person's meetings include all `whole-company` meetings plus all meetings of that person's division. Fill in the body of Alyssa's rule.

```
rule(meeting_time(person, day_and_time),
     rule-body);
```

c. Alyssa arrives at work on Wednesday morning and wonders what meetings she has to attend that day. Having defined the above rule, what query should she make to find this out?

Exercise 4.51

By giving the query

```
lives_near(person, list("Hacker", "Alyssa", "P"));
```

Alyssa P. Hacker is able to find people who live near her, with whom she can ride to work. On the other hand, when she tries to find all pairs of people who live near each other by querying

```
lives_near(person_1, person_2)
```

she notices that each pair of people who live near each other is listed twice; for example, Why does this happen? Is there a way to find a list of people who live near each other, in which each pair appears only once? Explain.

Logic as programs

We can regard a rule as a kind of logical implication: *If* an assignment of values to pattern variables satisfies the body, *then* it satisfies the conclusion. Consequently, we can regard the query language as having the ability to perform *logical deductions* based upon the rules. As an example, consider the append operation described at the beginning of section 4.4. As we said, append can be characterized by the following two rules:

- For any list `y`, the empty list and `y` append to form `y`.
- For any `u`, `v`, `y`, and `z`, `pair(u, v)` and `y` append to form `pair(u, z)` if `v` and `y` append to form `z`.

To express this in our query language, we define two rules for a relation

```
append-to-form("x", "y", "z")
```

which we can interpret to mean 'x and y append to form z':

```
rule(append_to_form(null, y, y));
rule(append_to_form(pair(u, v), y, pair(u, z)),
     append-to-form(v, y, z));
```

The first rule has no body, which means that the conclusion holds for any value of y. Note how the second rule makes use of dotted-tail notation to name the head and `tail` of a list.

Given these two rules, we can formulate queries that compute the append of two lists:

```
// Query input:
append-to-form(list("a", "b"), list("c", "d"), z);
// Query results:
append-to-form(list("a", "b"), list("c", "d"), list("a", "b", "c", "d"))
```

What is more striking, we can use the same rules to ask the question 'Which list, when appended to `list("a", "b")`, yields `list("a", "b", "c", "d")`?' This is done as follows:

```
// Query input:
append-to-form(list("a", "b"), y, list("a", "b", "c", "d"));
// Query results:
append-to-form(list("a", "b"), list("c", "d"), list("a", "b", "c", "d"))
```

We can also ask for all pairs of lists that append to form `list("a", "b", "c", "d")`:

```
// Query input:
append-to-form(x, y, list("a", "b", "c", "d"));
// Query results:
append-to-form(null, list("a", "b", "c", "d"), list("a", "b", "c", "d"))
append-to-form(list("a"), list("b", "c", "d"), list("a", "b", "c", "d"))
append-to-form(list("a", "b"), list("c", "d"), list("a", "b", "c", "d"))
append-to-form(list("a", "b", "c"), list("d"), list("a", "b", "c", "d"))
append-to-form(list("a", "b", "c", "d"), null, list("a", "b", "c", "d"))
```

The query system may seem to exhibit quite a bit of intelligence in using the rules to deduce the answers to the queries above. Actually, as we will see in the next section, the system is following a well-determined algorithm in unraveling the rules. Unfortunately, although the system works impressively in the append case, the general methods may break down in more complex cases, as we will see in section 4.4.3.

Exercise 4.52

The following rules implement a `next-to` relation that finds adjacent elements of a list:

```
rule(next_to_in(x, y, pair(x, pair(y, u))),
     and(next_to_in(x, y, pair(v, z)),
         next_to_in(x, y, z));
```

What will the response be to the following queries?

```
next_to_in(x, y, list(1, list(2, 3), 4));

next_to_in(x, 1, list(2, 1, 3, 1));
```

Exercise 4.53

Define rules to implement the `last_pair` operation of exercise 2.1, which returns a list containing the last element of a nonempty list. Check your rules on queries such as `last_pair(list(3), x)`, `last_pair(list(1, 2, 3), x)`, and `last_pair(list(2, x, list(3))`. Do your rules work correctly on queries such as `last_pair(x, list(3))`?

Exercise 4.54

The following data base (see Genesis 4) traces the genealogy of the descendants of Ada back to Adam, by way of Cain:

```
son("Adam", "Cain");
son("Cain", "Enoch");
son("Enoch", "Irad");
son("Methushael", "Lamech");
wife("Lamech", "Ada");
son("Ada", "Jabal");
son("Ada", "Jubal");
```

Formulate rules such as 'If *S* is the son of *F*, and *F* is the son of *G*, then *S* is the grandson of *G*' and 'If *W* is the wife of *M*, and *S* is the son of *W*, then *S* is the son of *M*' (which was supposedly more true in biblical times than today) that will enable the query system to find the grandson of Cain; the sons of Lamech; the grandsons of Methushael. (See exercise 4.60 for some rules to deduce more complicated relationships.)

4.4.2 How the Query System Works

In section 4.4.4 we will present an implementation of the query interpreter as a collection of functions. In this section we give an overview that explains the general structure of the system independent of low-level implementation details. After describing the implementation of the interpreter, we will be in a position to understand some of its limitations and some of the subtle ways in which the query language's logical operations differ from the operations of mathematical logic.

It should be apparent that the query evaluator must perform some kind of search in order to match queries against facts and rules in the data base. One way to do this would be to implement the query system as a nondeterministic program, using the `amb` evaluator of section 4.3 (see exercise 4.69). Another possibility is to manage the search with the aid of streams. Our implementation follows this second approach.

The query system is organized around two central operations called *pattern matching* and *unification*. We first describe pattern matching and explain how this operation, together with the organization of information in terms of streams of frames, enables us to implement both simple and compound queries. We next discuss unification, a generalization of pattern matching needed to implement rules. Finally, we show how the entire query interpreter fits together through a function that classifies expressions in a manner analogous to the way `evaluate` classifies expressions for the interpreter described in section 4.1.

Pattern matching

A *pattern matcher* is a program that tests whether some datum fits a specified pattern. For example, the data list `list(list("a", "b"), "c", list("a", "b"))` matches the pattern `list(x, "c", x)` with the pattern variable `x` bound to `list("a", "b")`. The same data list matches the pattern `list(x, y, z)` with `x` and `z` both bound to `list("a", "b")` and `y` bound to `"c"`. It also matches the pattern `list(list(x, y), "c", list(x, y))` with `x` bound to `"a"` and `y` bound to `"b"`. However, it does not match the pattern `list(x, "a", y)`, since that pattern specifies a list whose second element is the string `"a"`.

The pattern matcher used by the query system takes as inputs a pattern, a datum, and a *frame* that specifies bindings for various pattern variables. It checks whether the datum matches the pattern in a way that is consistent with the bindings already in the frame. If so, it returns the given frame augmented by any bindings that may have been determined by the match. Otherwise, it indicates that the match has failed.

For example, using the pattern `list(x, y, z)` to match `list("a", "b", "c")` given an empty frame will return a frame specifying that `x` is bound to `"a"` and `y` is bound to `"b"`. Trying the match with the same pattern, the same datum, and a frame specifying that `y` is bound to `"a"`

will fail. Trying the match with the same pattern, the same datum, and a frame in which y is bound to b and x is unbound will return the given frame augmented by a binding of x to "a".

The pattern matcher is all the mechanism that is needed to process simple queries that don't involve rules. For instance, to process the query

```
job(x, list("computer", "programmer"));
```

we scan through all assertions in the data base and select those that match the pattern with respect to an initially empty frame. For each match we find, we use the frame returned by the match to instantiate the pattern with a value for x.

Streams of frames

The testing of patterns against frames is organized through the use of streams. Given a single frame, the matching process runs through the data-base entries one by one. For each data-base entry, the matcher generates either a special symbol indicating that the match has failed or an extension to the frame. The results for all the data-base entries are collected into a stream, which is passed through a filter to weed out the failures. The result is a stream of all the frames that extend the given frame via a match to some assertion in the data base.[57]

In our system, a query takes an input stream of frames and performs the above matching operation for every frame in the stream, as indicated in Figure 4.4. That is, for each frame in the input stream, the query generates a new stream consisting of all extensions to that frame by matches to assertions in the data base. All these streams are then combined to form one huge stream, which contains all possible extensions of every frame in the input stream. This stream is the output of the query.

[57] Because matching is generally very expensive, we would like to avoid applying the full matcher to every element of the data base. This is usually arranged by breaking up the process into a fast, coarse match and the final match. The coarse match filters the data base to produce a small set of candidates for the final match. With care, we can arrange our data base so that some of the work of coarse matching can be done when the data base is constructed rather then when we want to select the candidates. This is called *indexing* the data base. There is a vast technology built around data-base-indexing schemes. Our implementation, described in section 4.4.4, contains a simple-minded form of such an optimization.

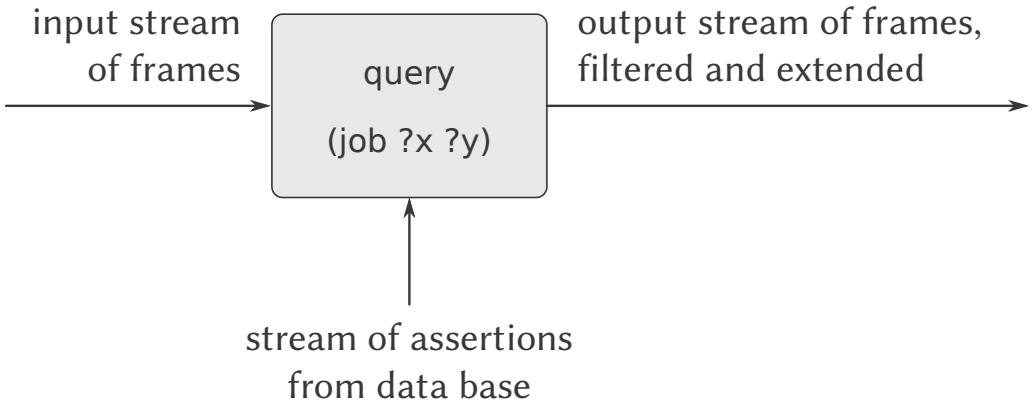


Figure 4.4: A query processes a stream of frames.

To answer a simple query, we use the query with an input stream consisting of a single empty frame. The resulting output stream contains all extensions to the empty frame (that is, all answers to our query). This stream of frames is then used to generate a stream of copies of the original query pattern with the variables instantiated by the values in each frame, and this is the stream that is finally printed.

Compound queries

The real elegance of the stream-of-frames implementation is evident when we deal with compound queries. The processing of compound queries makes use of the ability of our matcher to demand that a match be consistent with a specified frame. For example, to handle the `and` of two queries, such as

```
and(can_do_job(x, list("computer", "programmer", "trainee")),
    job(person, x))
```

(informally, 'Find all people who can do the job of a computer programmer trainee'), we first find all entries that match the pattern

```
can_do_job(x, list("computer", "programmer", "trainee"))
```

This produces a stream of frames, each of which contains a binding for `x`. Then for each frame in the stream we find all entries that match

```
job(person, x)
```

in a way that is consistent with the given binding for `x`. Each such match will produce a frame containing bindings for `x` and `person`. The `and` of two queries can be viewed as a series combination of the two component queries, as shown in figure 4.5. The frames that pass through the first query filter are filtered and further extended by the second query.

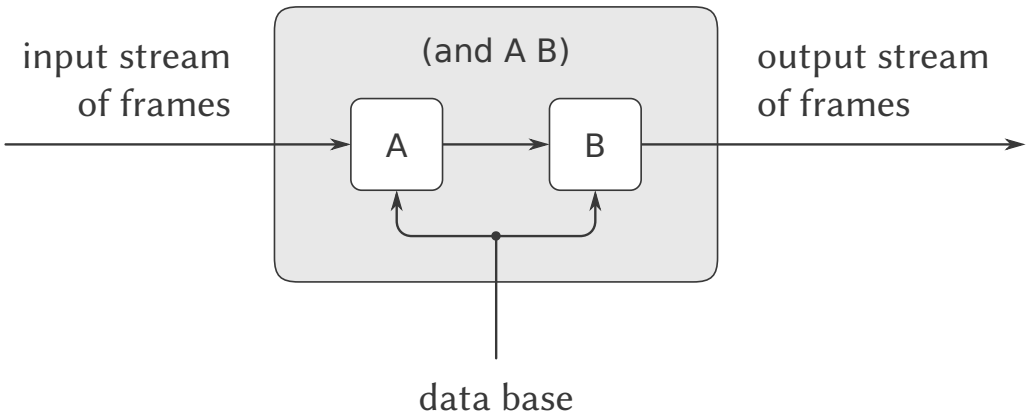


Figure 4.5: The and combination of two queries is produced by operating on the stream of frames in series.

Figure 4.6 shows the analogous method for computing the or of two queries as a parallel combination of the two component queries. The input stream of frames is extended separately by each query. The two resulting streams are then merged to produce the final output stream.

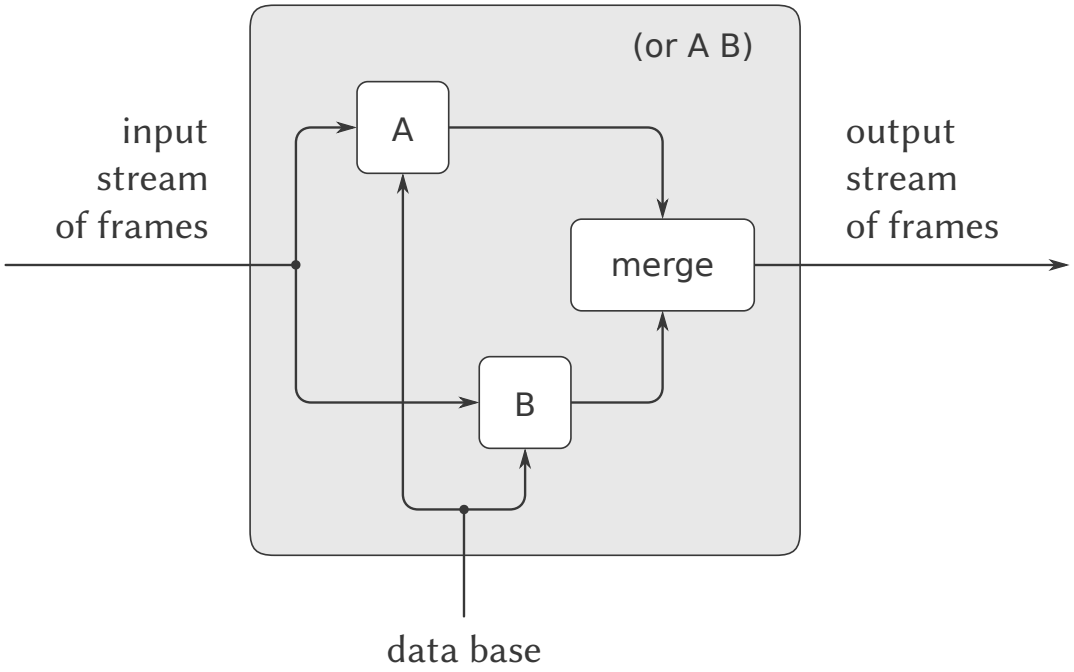


Figure 4.6: The or combination of two queries is produced by operating on the stream of frames in parallel and merging the results.

Even from this high-level description, it is apparent that the processing of compound queries can be slow. For example, since a query may produce more than one output frame for each input frame, and each query in an and gets its input frames from the previous query, an and query could, in the worst case, have to perform a number of matches that is exponential in the number of queries (see exercise 4.67).[58] Though systems for handling only simple queries

[58]But this kind of exponential explosion is not common in and queries because the added conditions tend to

are quite practical, dealing with complex queries is extremely difficult.[59]

From the stream-of-frames viewpoint, the `not` of some query acts as a filter that removes all frames for which the query can be satisfied. For instance, given the pattern

```
not(job(x, list("computer", "programmer")))
```

we attempt, for each frame in the input stream, to produce extension frames that satisfy `job(x, list("computer", "programmer"))`. We remove from the input stream all frames for which such extensions exist. The result is a stream consisting of only those frames in which the binding for `x` does not satisfy `job(x, list("computer", "programmer"))`. For example, in processing the query

```
and(supervisor(x, y),
    not(job(x, list("computer", "programmer"))));
```

the first clause will generate frames with bindings for `x` and `y`. The `not` clause will then filter these by removing all frames in which the binding for `x` satisfies the restriction that `x` is a computer programmer.[60]

The `javascript-value` expression is implemented as a similar filter on frame streams. We use each frame in the stream to instantiate any variables in the pattern, then apply the JavaScript predicate. We remove from the input stream all frames for which the predicate fails.

Unification

In order to handle rules in the query language, we must be able to find the rules whose conclusions match a given query pattern. Rule conclusions are like assertions except that they can contain variables, so we will need a generalization of pattern matching—called *unification*—in which both the 'pattern' and the 'datum' may contain variables.

A unifier takes two patterns, each containing constants and variables, and determines whether it is possible to assign values to the variables that will make the two patterns equal. If so, it returns a frame containing these bindings. For example, unifying `list(x, "a", y)` and `list(y, z, "a")` will specify a frame in which `x`, `y`, and `z` must all be bound to `"a"`. On the other hand, unifying `list(x, y, "a")` and `list(x, "b", y)` will fail, because there is no value for `y` that can make the two patterns equal. (For the second elements of the patterns to be equal, `y` would have to be `"b"`; however, for the third elements to be equal, `y` would have to be `"a"`.) The unifier used in the query system, like the pattern matcher, takes a frame as input and performs unifications that are consistent with this frame.

reduce rather than expand the number of frames produced.

[59] There is a large literature on data-base-management systems that is concerned with how to handle complex queries efficiently.

[60] There is a subtle difference between this filter implementation of `not` and the usual meaning of `not` in mathematical logic. See section 4.4.3.

The unification algorithm is the most technically difficult part of the query system. With complex patterns, performing unification may seem to require deduction. To unify `list(x, x)` and `list(list("a", y, "c"), list("a", "b", z))`, for example, the algorithm must infer that `x` should be `list("a", "b", "c")`, `y` should be `"b"`, and `z` should be `"c"`. We may think of this process as solving a set of equations among the pattern components. In general, these are simultaneous equations, which may require substantial manipulation to solve.[61] For example, unifying `list(x, x)` and `list(list("a", y, "c"), list("a", "b", z))` may be thought of as specifying the simultaneous equations

```
x = list("a", y, "c")
x = list("a", "b", z)
```

These equations imply that

```
list("a", y, "c") = list("a", "b", z)
```

which in turn implies that

```
"a" = "a", y = "b", "c" = z,
```

and hence that

```
x = list("a", "b", "c")
```

In a successful pattern match, all pattern variables become bound, and the values to which they are bound contain only constants. This is also true of all the examples of unification we have seen so far. In general, however, a successful unification may not completely determine the variable values; some variables may remain unbound and others may be bound to values that contain variables.

Consider the unification of `list(x, "a")` and `list(list("b", y), z)`. We can deduce that `x` = `list("b", y)` and `"a"` = `z`, but we cannot further solve for `x` or `y`. The unification doesn't fail, since it is certainly possible to make the two patterns equal by assigning values to `x` and `y`. Since this match in no way restricts the values `y` can take on, no binding for `y` is put into the result frame. The match does, however, restrict the value of `x`. Whatever value `y` has, `x` must be `list("b", y)`. A binding of `x` to the pattern `list("b", y)` is thus put into the frame. If a value for `y` is later determined and added to the frame (by a pattern match or unification that is required to be consistent with this frame), the previously bound `x` will refer to this value.[62]

[61] In one-sided pattern matching, all the equations that contain pattern variables are explicit and already solved for the unknown (the pattern variable).

[62] Another way to think of unification is that it generates the most general pattern that is a specialization of the two input patterns. That is, the unification of `list(x, "a")` and `list(list("b", y), z)` is `list(list("b", y), "a")`, and the unification of `list(x, "a", y)` and `list(y, z, "a")`, discussed above, is `list("a", "a", "a")`. For our implementation, it is more convenient to think of the result of unification as a frame rather than a pattern.

Applying rules

Unification is the key to the component of the query system that makes inferences from rules. To see how this is accomplished, consider processing a query that involves applying a rule, such as

```
lives_near(x, list("Hacker", "Alyssa", "P"));
```

To process this query, we first use the ordinary pattern-match function described above to see if there are any assertions in the data base that match this pattern. (There will not be any in this case, since our data base includes no direct assertions about who lives near whom.) The next step is to attempt to unify the query pattern with the conclusion of each rule. We find that the pattern unifies with the conclusion of the rule

```
rule(lives_near(person_1, person_2),
     and(address(person_1, pair(town, rest_1)),
         address(person_2, list(town, rest_2)),
         not(same(person_1, person_2))));
```

resulting in a frame specifying that `person_2` is bound to `list("Hacker", "Alyssa", "P")` and that `x` should be bound to (have the same value as) `person_1`. Now, relative to this frame, we evaluate the compound query given by the body of the rule. Successful matches will extend this frame by providing a binding for `person_1`, and consequently a value for `x`, which we can use to instantiate the original query pattern.

In general, the query evaluator uses the following method to apply a rule when trying to establish a query pattern in a frame that specifies bindings for some of the pattern variables:

- Unify the query with the conclusion of the rule to form, if successful, an extension of the original frame.
- Relative to the extended frame, evaluate the query formed by the body of the rule.

Notice how similar this is to the method for applying a function in the `evaluate`/`apply` evaluator for JavaScript:

- Bind the function's parameters to its arguments to form a frame that extends the original function environment.
- Relative to the extended environment, evaluate the expression formed by the body of the function.

The similarity between the two evaluators should come as no surprise. Just as function definitions are the means of abstraction in JavaScript, rule definitions are the means of abstraction in the query language. In each case, we unwind the abstraction by creating appropriate bindings and evaluating the rule or function body relative to these.

Simple queries

We saw earlier in this section how to evaluate simple queries in the absence of rules. Now that we have seen how to apply rules, we can describe how to evaluate simple queries by using both rules and assertions.

Given the query pattern and a stream of frames, we produce, for each frame in the input stream, two streams:

- a stream of extended frames obtained by matching the pattern against all assertions in the data base (using the pattern matcher), and
- a stream of extended frames obtained by applying all possible rules (using the unifier).[63]

Appending these two streams produces a stream that consists of all the ways that the given pattern can be satisfied consistent with the original frame. These streams (one for each frame in the input stream) are now all combined to form one large stream, which therefore consists of all the ways that any of the frames in the original input stream can be extended to produce a match with the given pattern.

The query evaluator and the driver loop

Despite the complexity of the underlying matching operations, the system is organized much like an evaluator for any language. The function that coordinates the matching operations is called `qeval`, and it plays a role analogous to that of the `evaluate` function for JavaScript. The function `qeval` takes as inputs a query and a stream of frames. Its output is a stream of frames, corresponding to successful matches to the query pattern, that extend some frame in the input stream, as indicated in Figure 4.4. Like `evaluate`, `qeval` classifies the different types of expressions (queries) and dispatches to an appropriate function for each. There is a function for each special form (`and`, `or`, `not`, and `javascript_value`) and one for simple queries.

The driver loop, which is analogous to the `driver_loop` function for the other evaluators in this chapter, reads queries from the terminal. For each query, it calls `qeval` with the query and a stream that consists of a single empty frame. This will produce the stream of all possible matches (all possible extensions to the empty frame). For each frame in the resulting stream, it instantiates the original query using the values of the variables found in the frame. This stream of instantiated queries is then printed.[64]

[63] Since unification is a generalization of matching, we could simplify the system by using the unifier to produce both streams. Treating the easy case with the simple matcher, however, illustrates how matching (as opposed to full-blown unification) can be useful in its own right.

[64] The reason we use streams (rather than lists) of frames is that the recursive application of rules can generate infinite numbers of values that satisfy a query. The delayed evaluation embodied in streams is crucial here: The system will print responses one by one as they are generated, regardless of whether there are a finite or infinite number of responses.

The driver also checks for the special command `assert`, which signals that the input is not a query but rather an assertion or rule to be added to the data base. For instance,

```
assert(job(list("Bitdiddle", "Ben"), list("computer", "wizard")));

assert(rule(wheel(person),
            and(supervisor(middle-manager, person),
                supervisor(x, middle-manager))));
```

4.4.3 Is Logic Programming Mathematical Logic?

The means of combination used in the query language may at first seem identical to the operations `and`, `or`, and `not` of mathematical logic, and the application of query-language rules is in fact accomplished through a legitimate method of inference.[65] This identification of the query language with mathematical logic is not really valid, though, because the query language provides a *control structure* that interprets the logical statements procedurally. We can often take advantage of this control structure. For example, to find all of the supervisors of programmers we could formulate a query in either of two logically equivalent forms:

```
and(job(x, list("computer", "programmer")),
    supervisor(x, y));
```

or

```
and(supervisor(x, y),
    job(x, list("computer", "programmer")));
```

If a company has many more supervisors than programmers (the usual case), it is better to use the first form rather than the second because the data base must be scanned for each intermediate result (frame) produced by the first clause of the `and`.

The aim of logic programming is to provide the programmer with techniques for decomposing a computational problem into two separate problems: 'what' is to be computed, and 'how' this should be computed. This is accomplished by selecting a subset of the statements of mathematical logic that is powerful enough to be able to describe anything one might want to compute, yet weak enough to have a controllable procedural interpretation. The intention here is that, on the one hand, a program specified in a logic programming language should be an effective program that can be carried out by a computer. Control ('how' to compute) is effected by using the order of evaluation of the language. We should be able to arrange the order of clauses and the order of subgoals within each clause so that the computation is done

[65] That a particular method of inference is legitimate is not a trivial assertion. One must prove that if one starts with true premises, only true conclusions can be derived. The method of inference represented by rule applications is *modus ponens*, the familiar method of inference that says that if A is true and A *implies* B is true, then we may conclude that B is true.

in an order deemed to be effective and efficient. At the same time, we should be able to view the result of the computation ('what' to compute) as a simple consequence of the laws of logic.

Our query language can be regarded as just such a procedurally interpretable subset of mathematical logic. An assertion represents a simple fact (an atomic proposition). A rule represents the implication that the rule conclusion holds for those cases where the rule body holds. A rule has a natural procedural interpretation: To establish the conclusion of the rule, establish the body of the rule. Rules, therefore, specify computations. However, because rules can also be regarded as statements of mathematical logic, we can justify any 'inference' accomplished by a logic program by asserting that the same result could be obtained by working entirely within mathematical logic.[66]

Infinite loops

A consequence of the procedural interpretation of logic programs is that it is possible to construct hopelessly inefficient programs for solving certain problems. An extreme case of inefficiency occurs when the system falls into infinite loops in making deductions. As a simple example, suppose we are setting up a data base of famous marriages, including

```
assert(married("Minnie", "Mickey"));
```

If we now ask

```
married("Mickey", who);
```

we will get no response, because the system doesn't know that if A is married to B, then B is married to A. So we assert the rule

```
assert(rule(married(x, y), married(y, x)));
```

and again query

```
married("Mickey", who);
```

Unfortunately, this will drive the system into an infinite loop, as follows:

- The system finds that the `married` rule is applicable; that is, the rule conclusion `married(x, y)` successfully unifies with the query pattern `married("Mickey", who)` to produce a frame

[66] We must qualify this statement by agreeing that, in speaking of the 'inference' accomplished by a logic program, we assume that the computation terminates. Unfortunately, even this qualified statement is false for our implementation of the query language (and also false for programs in Prolog and most other current logic programming languages) because of our use of `not` and `javascript_value`. As we will describe below, the `not` implemented in the query language is not always consistent with the `not` of mathematical logic, and `javascript_value` introduces additional complications. We could implement a language consistent with mathematical logic by simply removing `not` and `javascript_value` from the language and agreeing to write programs using only simple queries, `and`, and `or`. However, this would greatly restrict the expressive power of the language. One of the major concerns of research in logic programming is to find ways to achieve more consistency with mathematical logic without unduly sacrificing expressive power.

in which `x` is bound to `"Mickey"` and `y` is bound to `who`. So the interpreter proceeds to evaluate the rule body `married(x, y)` in this frame—in effect, to process the query `married(who, "Mickey")`.

- One answer appears directly as an assertion in the data base: `married("Minnie", "Mickey")`.
- The `married` rule is also applicable, so the interpreter again evaluates the rule body, which this time is equivalent to `married("Mickey", who)`.

The system is now in an infinite loop. Indeed, whether the system will find the simple answer `married("Minnie", "Mickey")` before it goes into the loop depends on implementation details concerning the order in which the system checks the items in the data base. This is a very simple example of the kinds of loops that can occur. Collections of interrelated rules can lead to loops that are much harder to anticipate, and the appearance of a loop can depend on the order of clauses in an `and` (see exercise 4.55) or on low-level details concerning the order in which the system processes queries.[67]

Problems with `not`

Another quirk in the query system concerns `not`. Given the data base of section 4.4.1, consider the following two queries:

```
and(supervisor(x, y),
    not(job(x, list("computer", "programmer"))));

and(not(job(x, list("computer", "programmer"))),
    supervisor(x, y));
```

These two queries do not produce the same result. The first query begins by finding all entries in the data base that match `supervisor(x, y)`, and then filters the resulting frames by removing the ones in which the value of `x` satisfies `job(x, list("computer", "programmer"))`. The second query begins by filtering the incoming frames to remove those that can satisfy `job(x, list("computer", "programmer"))`. Since the only incoming frame is empty, it checks the data base to see if there are any patterns that satisfy `job(x, list("computer", "programmer"))`. Since there generally are entries of this form, the `not` clause filters out the empty frame and re-

[67] This is not a problem of the logic but one of the procedural interpretation of the logic provided by our interpreter. We could write an interpreter that would not fall into a loop here. For example, we could enumerate all the proofs derivable from our assertions and our rules in a breadth-first rather than a depth-first order. However, such a system makes it more difficult to take advantage of the order of deductions in our programs. One attempt to build sophisticated control into such a program is described in deKleer et al. 1977. Another technique, which does not lead to such serious control problems, is to put in special knowledge, such as detectors for particular kinds of loops (exercise 4.58). However, there can be no general scheme for reliably preventing a system from going down infinite paths in performing deductions. Imagine a diabolical rule of the form 'To show $P(x)$ is true, show that $P(f(x))$ is true,' for some suitably chosen function f.

turns an empty stream of frames. Consequently, the entire compound query returns an empty stream.

The trouble is that our implementation of `not` really is meant to serve as a filter on values for the variables. If a `not` clause is processed with a frame in which some of the variables remain unbound (as does `x` in the example above), the system will produce unexpected results. Similar problems occur with the use of `javascript_value`—the JavaScript predicate can't work if some of its arguments are unbound. See exercise 4.68.

There is also a much more serious way in which the `not` of the query language differs from the `not` of mathematical logic. In logic, we interpret the statement 'not P' to mean that P is not true. In the query system, however, 'not P' means that P is not deducible from the knowledge in the data base. For example, given the personnel data base of section 4.4.1, the system would happily deduce all sorts of `not` statements, such as that Ben Bitdiddle is not a baseball fan, that it is not raining outside, and that $2 + 2$ is not 4.[68] In other words, the `not` of logic programming languages reflects the so-called *closed world assumption* that all relevant information has been included in the data base.[69]

Exercise 4.55

Louis Reasoner mistakenly deletes the `outranked_by` rule (section 4.4.1) from the data base. When he realizes this, he quickly reinstalls it. Unfortunately, he makes a slight change in the rule, and types it in as

```
rule(outranked_by(staff_person, boss),
     or(supervisor(staff_person, boss),
        and(outranked_by(middle_manager, boss),
            supervisor(staff_person, middle_manager))));
```

Just after Louis types this information into the system, DeWitt Aull comes by to find out who outranks Ben Bitdiddle. He issues the query

```
outanked_by(list("Bitdiddle", "Ben"), who);
```

After answering, the system goes into an infinite loop. Explain why.

Exercise 4.56

Cy D. Fect, looking forward to the day when he will rise in the organization, gives a query to find all the wheels (using the `wheel` rule of section 4.4.1):

[68] Consider the query `not(baseball_fan(list("Bitdiddle", "Ben")))`. The system finds that `baseball_fan(list("Bitdiddle", "Ben"))` is not in the data base, so the empty frame does not satisfy the pattern and is not filtered out of the initial stream of frames. The result of the query is thus the empty frame, which is used to instantiate the input query to produce `not(baseball_fan(list("Bitdiddle", "Ben")))`.

[69] A discussion and justification of this treatment of `not` can be found in the article by Clark (1978).

```
wheel(who);
```

To his surprise, the system responds

```
// Query results:
wheel(list("Warbucks", "Oliver"))
wheel(list("Bitdiddle", "Ben"))
wheel(list("Warbucks", "Oliver"))
wheel(list("Warbucks", "Oliver"))
wheel(list("Warbucks", "Oliver"))
```

Why is Oliver Warbucks listed four times?

Exercise 4.57

Ben has been generalizing the query system to provide statistics about the company. For example, to find the total salaries of all the computer programmers one will be able to say

```
sum(amount,
    and(job(x, list("computer", "programmer")),
        salary(x, amount)));
```

In general, Ben's new system allows expressions of the form

```
accumulation_function(variable, query-pattern)
```

where `accumulation_function` can be things like `sum`, `average`, or `maximum`. Ben reasons that it should be a cinch to implement this. He will simply feed the query pattern to `qeval`. This will produce a stream of frames. He will then pass this stream through a mapping function that extracts the value of the designated variable from each frame in the stream and feed the resulting stream of values to the accumulation function. Just as Ben completes the implementation and is about to try it out, Cy walks by, still puzzling over the `wheel` query result in exercise 4.56. When Cy shows Ben the system's response, Ben groans, 'Oh, no, my simple accumulation scheme won't work!' What has Ben just realized? Outline a method he can use to salvage the situation.

Exercise 4.58

Devise a way to install a loop detector in the query system so as to avoid the kinds of simple loops illustrated in the text and in exercise 4.55. The general idea is that the system should maintain some sort of history of its current chain of deductions and should not begin processing a query that it is already working on. Describe what kind of information (patterns and frames) is included in this history, and how the check should be made. (After you study the details of the query-system implementation in section 4.4.4, you may want to modify the system to include your loop detector.)

Exercise 4.59

Define rules to implement the `reverse` operation of exercise 2.2, which returns a list containing the same elements as a given list in reverse order. (Hint: Use `append_to_form`. Can your rules answer both `reverse(list(1, 2, 3), x)` and `reverse(x, list(1, 2, 3)`?

Exercise 4.60

Beginning with the data base and the rules you formulated in exercise 4.54, devise a rule for adding 'greats' to a grandson relationship. This should enable the system to deduce that Irad is the great-grandson of Adam, or that Jabal and Jubal are the great-great-great-great-great-grandsons of Adam. (Hint: Represent the fact about Irad, for example, as `list(list("great", "grandson"), '` Write rules that determine if a list ends in the word `"grandson"`. Use this to express a rule that allows one to derive the relationship `list(pair("great", rel), x, y)`, where `rel` is a list ending in `"grandson"`.) Check your rules on queries such as `list(list("great", "grandson"), g, ggs)` and `list(relationship, "Adam", "Irad")`.

4.4.4 Implementing the Query System

Section 4.4.2 described how the query system works. Now we fill in the details by presenting a complete implementation of the system.

4.4.4.1 The Driver Loop and Instantiation

The driver loop for the query system repeatedly reads input expressions. If the expression is a rule or assertion to be added to the data base, then the information is added. Otherwise the expression is assumed to be a query. The driver passes this query to the evaluator `qeval` together with an initial frame stream consisting of a single empty frame. The result of the evaluation is a stream of frames generated by satisfying the query with variable values found in the data base. These frames are used to form a new stream consisting of copies of the original query in which the variables are instantiated with values supplied by the stream of frames, and this final stream is printed at the terminal:

```
const input_prompt = "// Query input:";
const output_prompt = "// Query results:";

function query_driver_loop() {
    const input = prompt(input_prompt);
    const q = query_syntax_process(parse(input));
    if (assertion_to_be_added(q)) {
        add_rule_or_assertion(add_assertion_body(q));
```

```
        display("Assertion added to data base.");
    } else {
        display(output_prompt);
        display_stream(
            stream_map(
                frame =>
                    instantiate(q, frame,
                                (v, f) =>
                                   contract_question_mark(v)),
                qeval(q, singleton_stream(null))));

    }
    query_driver_loop();
}
```

Here, as in the other evaluators in this chapter, we use an abstract syntax for the expressions of the query language. The implementation of the expression syntax, including the predicate `assertion_to_be_added` and the selector `add_assertion_body`, is given in section 4.4.4.7. The function `add_rule_or_assertion` is defined in section 4.4.4.5.

Before doing any processing on an input expression, the driver loop transforms it syntactically into a form that makes the processing more efficient. This involves changing the representation of pattern variables. When the query is instantiated, any variables that remain unbound are transformed back to the input representation before being printed. These transformations are performed by the two functions `query_syntax_process` and `contract_question_mark` (section 4.4.4.7).

To instantiate an expression, we copy it, replacing any variables in the expression by their values in a given frame. The values are themselves instantiated, since they could contain variables (for example, if `x` in `exp` is bound to `y` as the result of unification and `y` is in turn bound to 5). The action to take if a variable cannot be instantiated is given by a procedural argument to `instantiate`.

```
function instantiate(exp, frame, unbound_var_handler) {
    function copy(exp) {
        if (is_var(exp)) {
            const binding = binding_in_frame(exp, frame);
            if (! binding === undefined) {
                return unbound_var_handler(exp, frame);
            } else {
                return copy(binding_value(binding));
            }
        } else if (is_pair(exp)) {
            return pair(copy(head(exp)), copy(tail(exp)));
        } else {
            return exp;
        }
```

```
    }
    copy(exp);
}
```

The functions that manipulate bindings are defined in section 4.4.4.8.

4.4.4.2 The Evaluator

The `qeval` function, called by the `query_driver_loop`, is the basic evaluator of the query system. It takes as inputs a query and a stream of frames, and it returns a stream of extended frames. It identifies special forms by a data-directed dispatch using `get` and `put`, just as we did in implementing generic operations in chapter 2. Any query that is not identified as a special form is assumed to be a simple query, to be processed by `simple_query`.

```
function qeval(query, frame_stream) {
    const qfun = get(type(query), "qeval");
    return qfun === undefined
        ? simple_query(query, frame_stream)
        : qfun(contents(query), frame_stream);
}
```

The functions `type` and `contents`, defined in section 4.4.4.7, implement the abstract syntax of the expressions.

Simple queries

The `simple_query` function handles simple queries. It takes as arguments a simple query (a pattern) together with a stream of frames, and it returns the stream formed by extending each frame by all data-base matches of the query.

```
function simple_query(query_pattern, frame_stream) {
    return stream_flatmap(
        frame =>
            stream_append_delayed(find_assertions(query_pattern, frame),
                                  delay(apply_rules(query_pattern, frame))),
        frame_stream);
}
```

For each frame in the input stream, we use `find_assertions` (section 4.4.4.3) to match the pattern against all assertions in the data base, producing a stream of extended frames, and we use `apply_rules` (section 4.4.4.4) to apply all possible rules, producing another stream of extended frames. These two streams are combined (using `stream_append_delayed`, section 4.4.4.6) to make a stream of all the ways that the given pattern can be satisfied consistent with the original frame (see exercise 4.62). The streams for the individual input frames are combined

using `stream_flatmap` (section 4.4.4.6) to form one large stream of all the ways that any of the frames in the original input stream can be extended to produce a match with the given pattern.

Compound queries

`And` queries are handled as illustrated in Figure 4.5 by the `conjoin` function, which takes as inputs the conjuncts and the frame stream and returns the stream of extended frames. First, `conjoin` processes the stream of frames to find the stream of all possible frame extensions that satisfy the first query in the conjunction. Then, using this as the new frame stream, it recursively applies `conjoin` to the rest of the queries.

```
function conjoin(conjuncts, frame_stream) {
    return is_empty_conjunction(conjuncts)
        ? frame_stream
        : conjoin(rest_conjuncts(conjuncts),
                  qeval(first_conjunct(conjuncts),
                        frame_stream));
}
```

The expression

```
put("and", "qeval", conjoin);
```

sets up `qeval` to dispatch to `conjoin` when an `and` expression is encountered.

`Or` queries are handled similarly, as shown in Figure 4.6. The output streams for the various disjuncts of the `or` are computed separately and merged using the `interleave_delayed` function from section 4.4.4.6. (See exercises 4.62 and 4.63.)

```
function disjoin(disjuncts, frame_stream) {
    return is_empty_disjunction(disjuncts)
        ? null
        : interleave_delayed(
              qeval(first_disjunct(disjuncts), frame_stream),
              delay(disjoin(rest_disjuncts(disjuncts),
                            frame_stream)));
}
put("or", "qeval", disjoin);
```

The predicates and selectors for the syntax of conjuncts and disjuncts are given in section 4.4.4.7.

Filters

`Not` is handled by the method outlined in section 4.4.2. We attempt to extend each frame in the input stream to satisfy the query being negated, and we include a given frame in the output stream only if it cannot be extended.

```
function negate(operands, frame_stream) {
    return stream_flatmap(
              frame =>
                 is_null(qeval(negated_query(operands),
                               singleton_stream(frame)))
                 ? singleton_stream(frame)
                 : null,
              frame_stream);
}
put("not", "qeval", negate);
```

`Javascript_value` is a filter similar to `not`. Each frame in the stream is used to instantiate the variables in the pattern, the indicated predicate is applied, and the frames for which the predicate returns false are filtered out of the input stream. An error results if there are unbound pattern variables.

```
function javascript_value(call, frame_stream) {
    return stream_flatmap(
              frame =>
                 execute(instantiate(call, frame,
                           (v, f) =>
                               error(v, "Unknown pat var --- javascript_value")))
                 ? singleton_stream(frame)
                 : null,
              frame_stream);
}
put("javascript_value", "qeval", javascript_value);
```

The function `execute`, which applies the predicate to the arguments, must `evaluate` the predicate expression to get the function to apply. However, it must not evaluate the arguments, since they are already the actual arguments, not expressions whose evaluation (in JavaScript) will produce the arguments. Note that `execute` is implemented using `evaluate` and `apply` from the underlying JavaScript system.

```
function execute(exp) {
    return apply(evaluate(predicate(exp), user_initial_environment),
                 args(exp));
}
```

The `always_true` expression provides for a query that is always satisfied. It ignores its contents (normally empty) and simply passes through all the frames in the input stream. The

always_true expression is used by the rule_body selector (section 4.4.4.7) to provide bodies for rules that were defined without bodies (that is, rules whose bodies are always satisfied).

```
function always_true(ignore, frame_stream) {
    return frame_stream;
}
put("always_true", "qeval", always_true);
```

The selectors that define the syntax of not and javascript_value are given in section 4.4.4.7.

4.4.4.3 Finding Assertions by Pattern Matching

The function find_assertions, called by simple_query (section 4.4.4.2), takes as input a pattern and a frame. It returns a stream of frames, each extending the given one by a data-base match of the given pattern. It uses fetch_assertions (section 4.4.4.5) to get a stream of all the assertions in the data base that should be checked for a match against the pattern and the frame. The reason for fetch_assertions here is that we can often apply simple tests that will eliminate many of the entries in the data base from the pool of candidates for a successful match. The system would still work if we eliminated fetch_assertions and simply checked a stream of all assertions in the data base, but the computation would be less efficient because we would need to make many more calls to the matcher.

```
function find_assertions(pattern, frame) {
    return stream_flatmap(
              datum =>
                 check_an_assertion(datum, pattern, frame),
              fetch_assertions(pattern, frame));
}
```

The function check_an_assertion takes as arguments a pattern, a data object (assertion), and a frame and returns either a one-element stream containing the extended frame or null if the match fails.

```
function check_an_assertion(assertion, query_pat, query_frame) {
    const match_result = pattern_match(query_pat, assertion,
                                       query_frame);
    return match_result === "failed"
        ? null
        : singleton_stream(match_result);
}
```

The basic pattern matcher returns either the string "failed" or an extension of the given frame. The basic idea of the matcher is to check the pattern against the data, element by element, accumulating bindings for the pattern variables. If the pattern and the data object are the same, the match succeeds and we return the frame of bindings accumulated so far. Otherwise, if the

pattern is a variable we extend the current frame by binding the variable to the data, so long as this is consistent with the bindings already in the frame. If the pattern and the data are both pairs, we (recursively) match the head of the pattern against the head of the data to produce a frame; in this frame we then match the `tail` of the pattern against the `tail` of the data. If none of these cases are applicable, the match fails and we return the symbol `failed`.

```
function pattern_match(pat, dat, frame) {
    return frame === "failed"
        ? "failed"
        : equal(pat, dat)
        ? frame
        : is_var(pat)
        ? extend_if_consistent(pat, dat, frame)
        : is_pair(pat) && is_pair(dat)
        ? pattern_match(tail(pat),
                        tail(dat),
                        pattern_match(head(pat),
                                      head(dat),
                                      frame))
        : "failed";
}
```

Here is the function that extends a frame by adding a new binding, if this is consistent with the bindings already in the frame:

```
function extend_if_consistent(variable, dat, frame) {
    const binding = binding_in_frame(variable, frame);
    return binding !== undefined
        ? extend(variable, dat, frame)
        : pattern_match(binding_value(binding), dat, frame);
}
```

If there is no binding for the variable in the frame, we simply add the binding of the variable to the data. Otherwise we match, in the frame, the data against the value of the variable in the frame. If the stored value contains only constants, as it must if it was stored during pattern matching by `extend_if_consistent`, then the match simply tests whether the stored and new values are the same. If so, it returns the unmodified frame; if not, it returns a failure indication. The stored value may, however, contain pattern variables if it was stored during unification (see section 4.4.4.4). The recursive match of the stored pattern against the new data will add or check bindings for the variables in this pattern. For example, suppose we have a frame in which `x` is bound to `list("f", y)` and `y` is unbound, and we wish to augment this frame by a binding of `x` to `list("f", "b")`. We look up `x` and find that it is bound to `list("f", y)`. This leads us to match `list("f", y)` against the proposed new value `list("f", "b")` in the same frame. Eventually this match extends the frame by adding a binding of `y` to `"b"`. The variable `x` remains bound to `list("f", y)`. We never modify a stored binding and we never store more

than one binding for a given variable.

The functions used by `extend_if_consistent` to manipulate bindings are defined in section 4.4.4.8.

4.4.4.4 Rules and Unification

The function `apply_rules` is the rule analog of `find_assertions` (section 4.4.4.3). It takes as input a pattern and a frame, and it forms a stream of extension frames by applying rules from the data base. The function `stream_flatmap` maps `apply_a_rule` down the stream of possibly applicable rules (selected by `fetch_rules`, section 4.4.4.5) and combines the resulting streams of frames.

```
function apply_rules(pattern, frame) {
    return stream_flatmap(
              rule =>
                 apply_a_rule(rule, pattern, frame),
              fetch_rules(pattern, frame));
}
```

The function `apply_a_rule` applies rules using the method outlined in section 4.4.2. It first augments its argument frame by unifying the rule conclusion with the pattern in the given frame. If this succeeds, it evaluates the rule body in this new frame.

Before any of this happens, however, the program renames all the variables in the rule with unique new names. The reason for this is to prevent the variables for different rule applications from becoming confused with each other. For instance, if two rules both use a variable named `x`, then each one may add a binding for `x` to the frame when it is applied. These two `x`'s have nothing to do with each other, and we should not be fooled into thinking that the two bindings must be consistent. Rather than rename variables, we could devise a more clever environment structure; however, the renaming approach we have chosen here is the most straightforward, even if not the most efficient. (See exercise 4.70.) Here is the `apply_a_rule` function:

```
function apply_a_rule(rule, query_pattern, query_frame) {
    const clean_rule = rename_variables_in(rule);
    const unify_result =
             unify_match(query_pattern,
                         conclusion(clean_rule),
                         query_frame);
    return unify_result === "failed"
        ? null
        : qeval(rule_body(clean_rule),
                singleton_stream(unify_result));
}
```

The selectors `rule_body` and `conclusion` that extract parts of a rule are defined in section 4.4.4.7.

We generate unique variable names by associating a unique identifier (such as a number) with each rule application and combining this identifier with the original variable names. For example, if the rule-application identifier is 7, we might change each x in the rule to x_7 and each y in the rule to y_7. (The functions make_new_variable and new_rule_application_id are included with the syntax functions in section 4.4.4.7.)

```
function rename_variables_in(rule) {
    const rule_application_id = new_rule_application_id();
    function tree_walk(exp) {
        return is_var(exp)
            ? make_new_variable(exp, rule_application_id)
            : is_pair(exp)
            ? pair(tree_walk(head(exp)),
                   tree_walk(tail(exp)))
            : exp;
    }
    tree_walk(rule);
}
```

The unification algorithm is implemented as a function that takes as inputs two patterns and a frame and returns either the extended frame or the string "failed". The unifier is like the pattern matcher except that it is symmetrical—variables are allowed on both sides of the match. The function unify_match is basically the same as pattern_match, except that there is extra code (marked '***' below) to handle the case where the object on the right side of the match is a variable.

```
function unify_match(p1, p2, frame) {
    return frame === "failed"
        ? "failed"
        : equal(p1, p2)
        ? frame
        : is_var(p1)
        ? extend_if_possible(p1, p2, frame)
        : is_var(p2)
        ? extend_if_possible(p2, p1, frame)  // ***
        : pair(p1) && pair(p2)
        ? unify_match(tail(p1),
                      tail(p2),
                      unify_match(head(p1),
                                  head(p2),
                                  frame))
        : "failed";
}
```

In unification, as in one-sided pattern matching, we want to accept a proposed extension of the frame only if it is consistent with existing bindings. The function extend_if_possible used in unification is the same as the extend_if_consistent used in pattern matching except for

two special checks, marked '***' in the program below. In the first case, if the variable we are trying to match is not bound, but the value we are trying to match it with is itself a (different) variable, it is necessary to check to see if the value is bound, and if so, to match its value. If both parties to the match are unbound, we may bind either to the other.

The second check deals with attempts to bind a variable to a pattern that includes that variable. Such a situation can occur whenever a variable is repeated in both patterns. Consider, for example, unifying the two patterns `list(x, x)` and `list(y, `*expression involving* `y)` in a frame where both `x` and `y` are unbound. First `x` is matched against `y`, making a binding of `x` to `y`. Next, the same `x` is matched against the given expression involving `y`. Since `x` is already bound to `y`, this results in matching `y` against the expression. If we think of the unifier as finding a set of values for the pattern variables that make the patterns the same, then these patterns imply instructions to find a `y` such that `y` is equal to the *expression involving*`y`. There is no general method for solving such equations, so we reject such bindings; these cases are recognized by the predicate `depends_on`.[70] On the other hand, we do not want to reject attempts to bind a variable to itself. For example, consider unifying `list(x, x)` and `list(y, y)`. The second attempt to bind `x` to `y` matches `y` (the stored value of `x`) against `y` (the new value of `x`). This is taken care of by the `equal` clause of `unify_match`.

```
function extend_if_possible(variable, val, frame) {
    const binding = binding_in_frame(variable, frame);
    if (binding !== undefined) {
        return unify_match(binding_value(binding),
                           val, frame);
    } else if (is_var(val)) {                        // ***
        const binding = binding_in_frame(val, frame);
```

[70] In general, unifying `y` with an expression involving `y` would require our being able to find a fixed point of the equation `y` = *expression involving*`y`. It is sometimes possible to syntactically form an expression that appears to be the solution. For example, `y = list("f", y)` seems to have the fixed point `list("f", list("f", list("f",...)))`, which we can produce by beginning with the expression `list("f", y)` and repeatedly substituting `list("f", y)` for `y`. Unfortunately, not every such equation has a meaningful fixed point. The issues that arise here are similar to the issues of manipulating infinite series in mathematics. For example, we know that 2 is the solution to the equation $y = 1 + y/2$. Beginning with the expression $1 + y/2$ and repeatedly substituting $1 + y/2$ for y gives

$$2 = y = 1 + y/2 = 1 + (1 + y/2)/2 = 1 + 1/2 + y/4 = \cdots ,$$

which leads to

$$2 = 1 + 1/2 + 1/4 + 1/8 + \cdots .$$

However, if we try the same manipulation beginning with the observation that -1 is the solution to the equation $y = 1 + 2y$, we obtain

$$-1 = y = 1 + 2y = 1 + 2(1 + 2y) = 1 + 2 + 4y = \cdots ,$$

which leads to

$$-1 = 1 + 2 + 4 + 8 + \cdots .$$

Although the formal manipulations used in deriving these two equations are identical, the first result is a valid assertion about infinite series but the second is not. Similarly, for our unification results, reasoning with an arbitrary syntactically constructed expression may lead to errors.

```
        return binding !== undefined
            ? unify_match(variable,
                          binding_value(binding),
                          frame)
            : extend(variable, val, frame);
    } else if (depends_on(val, variable, frame)) { // ***
        return "failed";
    } else {
        return extend(variable, val, frame);
    }
}
```

The function depends_on is a predicate that tests whether an expression proposed to be the value of a pattern variable depends on the variable. This must be done relative to the current frame because the expression may contain occurrences of a variable that already has a value that depends on our test variable. The structure of depends_on is a simple recursive tree walk in which we substitute for the values of variables whenever necessary.

```
function depends_on(exp, variable, frame) {
    function tree_walk(e) {
        if (is_var(e)) {
            if (equal(variable, e)) {
                return true;
            } else {
                const b = binding_in_frame(e, frame);
                if (b !== undefined) {
                    return tree_walk(binding_value(b));
                } else {
                    return false;
                }
            }
        } else if (is_pair(e)) {
            return tree_walk(head(e)) ||
                   tree_walk(tail(e));
        } else {
            return false;
        }
    }
    return tree_walk(exp);
}
```

4.4.4.5 Maintaining the Data Base

One important problem in designing logic programming languages is that of arranging things so that as few irrelevant data-base entries as possible will be examined in checking a given pattern. In our system, in addition to storing all assertions in one big stream, we store all assertions whose heads are constant symbols in separate streams, in a table indexed by the symbol. To fetch an assertion that may match a pattern, we first check to see if the head of the pattern is a constant symbol. If so, we return (to be tested using the matcher) all the stored assertions that have the same head. If the pattern's head is not a constant symbol, we return all the stored assertions. Cleverer methods could also take advantage of information in the frame, or try also to optimize the case where the head of the pattern is not a constant symbol. We avoid building our criteria for indexing (using the head, handling only the case of constant symbols) into the program; instead we call on predicates and selectors that embody our criteria.

```
let THE_ASSERTIONS = null;

function fetch_assertions(pattern, frame) {
    return use_index(pattern)
        ? get_indexed_assertions(pattern)
        : get_all_assertions;
}
function get_all_assertions() {
    return THE_ASSERTIONS;
}
function get_indexed_assertions(pattern) {
    return get_stream(index_key_of(pattern), "assertion-stream");
}
```

The function get_stream looks up a stream in the table and returns an empty stream if nothing is stored there.

```
function get_stream(key1, key2) {
    const s = get(key1, key2);
    return s !== undefined ? s : null;
}
```

Rules are stored similarly, using the head of the rule conclusion. Rule conclusions are arbitrary patterns, however, so they differ from assertions in that they can contain variables. A pattern whose head is a constant symbol can match rules whose conclusions start with a variable as well as rules whose conclusions have the same head. Thus, when fetching rules that might match a pattern whose head is a constant symbol we fetch all rules whose conclusions start with a variable as well as those whose conclusions have the same head as the pattern. For this purpose we store all rules whose conclusions start with a variable in a separate stream in our table, indexed by the string "variables".

```
let THE_RULES = null;
```

```
function fetch_rules(pattern, frame) {
    return use_index(pattern)
        ? get_indexed_rules(pattern)
        : get_all_rules();
}
function get_all_rules() {
    return THE_RULES;
}
function get_indexed_rules(pattern) {
    return stream_append(
              get_stream(index_key_of(pattern),
                         "rule-stream"),
              get_stream("variables", "rule-stream"));
}
```

The function add_rule_or_assertion is used by query_driver_loop to add assertions and rules to the data base. Each item is stored in the index, if appropriate, and in a stream of all assertions or rules in the data base.

```
function add_rule_or_assertion(assertion) {
    return is_rule(assertion)
        ? add_rule(assertion)
        : add_assertion(assertion);
}
function add_assertion(assertion) {
    store_assertion_in_index(assertion);
    const old_assertions = THE_ASSERTIONS;
    THE_ASSERTIONS = pair(assertion, () => old_assertions);
    return "ok";
}
function add_rule(rule) {
    store_rule_in_index(rule);
    const old_rules = THE_RULES;
    THE_RULES = pair(rule, () => old_rules);
    return "ok";
}
```

To actually store an assertion or a rule, we check to see if it can be indexed. If so, we store it in the appropriate stream.

```
function store_assertion_in_index(assertion) {
    if (is_indexable(assertion)) {
        const key = index_key_of(assertion);
        const current_assertion_stream =
                    get_stream(key, "assertion-stream");
        put(key, "assertion-stream",
            pair(assertion, () => current_assertion_stream));
    } else {
```

```
    }
}
function store_rule_in_index(rule) {
    const pattern = conclusion(rule);
    if (is_indexable(pattern)) {
        const key = index_key_of(pattern);
        const current_rule_stream =
                    get_stream(key, "rule-stream");
        put(key, "rule-stream",
            pair(rule, () => current_rule_stream));
    } else {
    }
}
```

The following functions define how the data-base index is used. A pattern (an assertion or a rule conclusion) will be stored in the table if it starts with a variable or a string.

```
function indexable(pat) {
    return is_string(pat) || is_var(head(pat));
}
```

The key under which a pattern is stored in the table is either `"variables"` (if it starts with a variable) or the string with which it starts.

```
function index_key_of(pat) {
    const key = head(pat);
    return is_var(key) ? "variables" : key;
}
```

The index will be used to retrieve items that might match a pattern if the pattern starts with a string.

```
function use_index(pat) {
    return is_string(head(pat));
}
```

Exercise 4.61

What is the purpose of the constant declarations in the functions `add_assertion` and `add_rule`? What would be wrong with the following implementation of `add_assertion`? Hint: Recall the definition of the infinite stream of ones in section 3.5.2: **`const`** `ones = pair(1, () => ones);`.

```
function add_assertion(assertion) {
    store_assertion_in_index(assertion);
    THE_ASSERTIONS = pair(assertion, () => THE_ASSERTIONS);
    return "ok";
}
```

4.4.4.6 Stream Operations

The query system uses a few stream operations that were not presented in chapter 3.

The functions `stream_append_delayed` and `interleave_delayed` are just like `stream_append` and `interleave` (section 3.5.3), except that they take a delayed argument (like the `integral` function in section 3.5.4). This postpones looping in some cases (see exercise 4.62).

```
function stream_append_delayed(s1, delayed_s2) {
    return is_null(s1)
        ? delayed_s2()
        : pair(head(s1),
               () => stream_append_delayed(stream_tail(s1),
                                           delayed_s2));
}
function interleave_delayed(s1, delayed_s2) {
    return is_null(s1)
        ? delayed_s2()
        : pair(head(s1),
               () => interleave_delayed(delayed_s2(),
                         () => stream_tail(s1)));
}
```

The function `stream_flatmap`, which is used throughout the query evaluator to map a function over a stream of frames and combine the resulting streams of frames, is the stream analog of the `flatmap` function introduced for ordinary lists in section 2.1.3. Unlike ordinary `flatmap`, however, we accumulate the streams with an interleaving process, rather than simply appending them (see exercises 4.63 and 4.64).

```
function stream_flatmap(fun, s) {
    return flatten_stream(stream_map(fun, s));
}
function flatten_stream(stream) {
    return is_null(stream)
        ? null
        : interleave_delayed(head(stream),
              () => flatten_stream(stream_tail(stream)));
}
```

The evaluator also uses the following simple function to generate a stream consisting of a single element:

```
function singleton_stream(x) {
    pair(x, () => null);
}
```

4.4.4.7 Query Syntax Functions

The functions `type` and `contents`, used by `qeval` (section 4.4.4.2), specify that a query expression is identified by name of its operator.

```
function type(exp) {
    return is_application(exp) && is_name(operator(exp))
        ? name_of_name(operator(exp))
        : error(exp, "Unknown expression TYPE");
}
function contents(exp) {
    return is_application(exp) && is_name(operator(exp))
        ? operands(exp)
        : error(exp, "Unknown expression CONTENTS");
}
```

The following functions, used by `query_driver_loop` (in section 4.4.4.1), specify that rules and assertions are added to the data base by expressions of the form `assert(`*rule-or-assertion*):

```
function assertion_to_be_added(exp) {
    return type(exp) === "assert";
}
function add_assertion_body(exp) {
    return head(contents(exp));
}
```

Here are the syntax definitions for the `and`, `or`, `not`, and `javascript_value` query expressions (section 4.4.4.2):

```
function empty_conjunction(exps) {
    return is_null(exps);
}
function first_conjunct(exps) {
    return head(exps);
}
function rest_conjuncts(exps) {
    return tail(exps);
}
function is_empty_disjunction(exps) {
    return is_null(exps);
}
function first_disjunct(exps) {
    return head(exps);
}
function rest_disjuncts(exps) {
    return tail(exps);
}
function negated_query(exps) {
    return head(exps);
```

```
}
function predicate(exps) {
    return head(exps);
}
function args(exps) {
    return tail(exps);
}
```

The following three functions define the syntax of rules:

```
function is_rule(statement) {
    return is_application(statement) &&
           is_name(operator(statement)) &&
           name_of_name(operator(statement)) === "rule";
}
function conclusion(rule) {
    return head(operands(rule));
}
function rule_body(rule) {
    return is_null(tail(operands(rule)))
        ? "always_true"
        : tail(operands(rule));
}
```

The function `query_driver_loop` (section 4.4.4.1) calls `query_syntax_process` to transform pattern variables in the expression, which are names, into the internal format `list("?", name)`. That is to say, a pattern such as `list(x, y)` is actually represented internally by the system as `list(list("?", "x"), list("?", "y"))`. The syntax transformation is accomplished by the following function:

```
function query_syntax_process(exp) {
    return map_over_names(expand_question_mark, exp);
}
function map_over_names(fun, exp) {
    return is_name(exp)
        ? fun(exp)
        : is_pair(exp)
        ? pair(map_over_names(fun, head(exp)),
               map_over_names(fun, tail(exp)))
        : exp;
}
function expand_question_mark(name) {
    return list("?", name_of_name(name));
}
```

Once the variables are transformed in this way, the variables in a pattern are lists starting with `"?"`, and the strings (which need to be recognized for data-base indexing, section 4.4.4.5) are just the strings.

```
function is_var(exp) {
    return is_tagged_list(exp, "?");
}
function is_constant_string(exp) {
    return is_string(exp);
}
```

Unique variables are constructed during rule application (in section 4.4.4.4) by means of the following functions. The unique identifier for a rule application is a number, which is incremented each time a rule is applied.

```
let rule_counter = 0;

function new_rule_application_id() {
    rule_counter = rule_counter + 1;
    return rule_counter;
}
function make_new_variable(variable, rule_application_id) {
    return list("?", tail(variable) + stringify(rule_application_id));
}
```

When query_driver_loop instantiates the query to print the answer, it converts any unbound pattern variables back to the right form for printing, using

```
function contract_question_mark(variable) {
    return list("name", tail(variable));
}
```

4.4.4.8 Frames and Bindings

Frames are represented as lists of bindings, which are variable-value pairs:

```
function make_binding(variable, value) {
    return pair(variable, value);
}
function binding_variable(binding) {
    return head(binding);
}
function binding_value(binding) {
    return tail(binding);
}
function binding_in_frame(variable, frame) {
    return assoc(variable, frame);
}
function extend(variable, value, frame) {
    return pair(make_binding(variable, value), frame);
}
```

Exercise 4.62

Louis Reasoner wonders why the `simple_query` and `disjoin` functions (section 4.4.4.2) are implemented using explicit delay operations, rather than being defined as follows:

```
function simple_query(query_pattern, frame_stream) {
    return stream_flatmap(
              frame =>
                 stream_append(find_assertions(query_pattern, frame),
                               apply_rules(query_pattern, frame)),
              frame_stream);
}
function disjoin(disjuncts, frame_stream) {
    return is_empty_disjunction(disjuncts)
        ? null
        : interleave(qeval(first_disjunct(disjuncts), frame_stream),
                     disjoin(rest_disjuncts(disjuncts), frame_stream));
}
```

Can you give examples of queries where these simpler definitions would lead to undesirable behavior?

Exercise 4.63

Why do `disjoin` and `stream_flatmap` interleave the streams rather than simply append them? Give examples that illustrate why interleaving works better. (Hint: Why did we use `interleave` in section 3.5.3?)

Exercise 4.64

Why does `flatten_stream` use a function definition expression in its body? What would be wrong with defining it as follows:

```
function flatten_stream(stream) {
    return is_null(stream)
        ? null
        : interleave(head(stream),
                     flatten_stream(stream_tail(stream)));
}
```

Exercise 4.65

Alyssa P. Hacker proposes to use a simpler version of `stream_flatmap` in `negate`, `javascript_value`, and `find_assertions`. She observes that the function that is mapped over the frame stream in

these cases always produces either the empty stream or a singleton stream, so no interleaving is needed when combining these streams.

a. Fill in the missing expressions in Alyssa's program.

```
function simple_stream_flatmap(fun, s) {
    return simple_flatten(stream_map(fun, s));
}
function simple_flatten(stream) {
    return stream_map(...,
                      stream_filter(..., stream));
}
```

b. Does the query system's behavior change if we change it in this way?

Exercise 4.66

Implement for the query language a query expression called `unique`. `Unique` should succeed if there is precisely one item in the data base satisfying a specified query. For example,

```
unique(job(x, list("computer", "wizard")));
```

should print the one-item stream

```
unique(job(list("Bitdiddle", "Ben"), list("computer", "wizard")))
```

since Ben is the only computer wizard, and

```
unique(job(x, list("computer", "programmer")));
```

should print the empty stream, since there is more than one computer programmer. Moreover,

```
and(job(x, j), unique(job(anyone, j)));
```

should list all the jobs that are filled by only one person, and the people who fill them. There are two parts to implementing `unique`. The first is to write a function that handles this special form, and the second is to make `qeval` dispatch to that function. The second part is trivial, since `qeval` does its dispatching in a data-directed way. If your function is called `uniquely_asserted`, all you need to do is

```
put("unique", "qeval", uniquely_asserted);
```

and `qeval` will dispatch to this function for every query whose `type` (`head`) is the nam `unique`. The real problem is to write the function `uniquely_asserted`. This should take as input the `contents` (`tail`) of the `unique` query, together with a stream of frames. For each frame in the stream, it should use `qeval` to find the stream of all extensions to the frame that satisfy the given query. Any stream that does not have exactly one item in it should be eliminated. The

remaining streams should be passed back to be accumulated into one big stream that is the result of the `unique` query. This is similar to the implementation of the `not` special form. Test your implementation by forming a query that lists all people who supervise precisely one person.

Exercise 4.67

Our implementation of `and` as a series combination of queries (figure 4.5) is elegant, but it is inefficient because in processing the second query of the `and` we must scan the data base for each frame produced by the first query. If the data base has N elements, and a typical query produces a number of output frames proportional to N (say N/k), then scanning the data base for each frame produced by the first query will require N^2/k calls to the pattern matcher. Another approach would be to process the two clauses of the `and` separately, then look for all pairs of output frames that are compatible. If each query produces N/k output frames, then this means that we must perform N^2/k^2 compatibility checks—a factor of k fewer than the number of matches required in our current method. Devise an implementation of `and` that uses this strategy. You must implement a function that takes two frames as inputs, checks whether the bindings in the frames are compatible, and, if so, produces a frame that merges the two sets of bindings. This operation is similar to unification.

Exercise 4.68

In section 4.4.3 we saw that `not` and `javascript_value` can cause the query language to give 'wrong' answers if these filtering operations are applied to frames in which variables are unbound. Devise a way to fix this shortcoming. One idea is to perform the filtering in a 'delayed' manner by appending to the frame a 'promise' to filter that is fulfilled only when enough variables have been bound to make the operation possible. We could wait to perform filtering until all other operations have been performed. However, for efficiency's sake, we would like to perform filtering as soon as possible so as to cut down on the number of intermediate frames generated.

Exercise 4.69

Redesign the query language as a nondeterministic program to be implemented using the evaluator of section 4.3, rather than as a stream process. In this approach, each query will produce a single answer (rather than the stream of all answers) and the user can type **`try`**`-again` to see more answers. You should find that much of the mechanism we built in this section is subsumed by nondeterministic search and backtracking. You will probably also find, however, that your new query language has subtle differences in behavior from the one implemented

here. Can you find examples that illustrate this difference?

Exercise 4.70

When we implemented the JavaScript evaluator in section 4.1, we saw how to use local environments to avoid name conflicts between the parameters of functions. For example, in evaluating

```
function square(x) {
    return x * x;
}
function sum_of_squares(x, y) {
    return square(x) + square(y);
}
sum_of_squares(3, 4);
```

there is no confusion between the `x` in `square` and the `x` in `sum_of_squares`, because we evaluate the body of each function in an environment that is specially constructed to contain bindings for the local variables. In the query system, we used a different strategy to avoid name conflicts in applying rules. Each time we apply a rule we rename the variables with new names that are guaranteed to be unique. The analogous strategy for the JavaScript evaluator would be to do away with local environments and simply rename the variables in the body of a function each time we apply the function. Implement for the query language a rule-application method that uses environments rather than renaming. See if you can build on your environment structure to create constructs in the query language for dealing with large systems, such as the rule analog of block-structured functions. Can you relate any of this to the problem of making deductions in a context (e.g., 'If I supposed that *P* were true, then I would be able to deduce *A* and *B*.') as a method of problem solving? (This problem is open-ended. A good answer is probably worth a Ph.D.)

Chapter 5

Computing with Register Machines

> My aim is to show that the heavenly machine is not a kind of divine, live being, but a kind of clockwork (and he who believes that a clock has soul attributes the maker's glory to the work), insofar as nearly all the manifold motions are caused by a most simple and material force, just as all motions of the clock are caused by a single weight.
>
> — Johannes Kepler ((letter to Herwart von Hohenburg, 1605))

We began this book by studying processes and by describing processes in terms of functions written in JavaScript. To explain the meanings of these functions, we used a succession of models of evaluation: the substitution model of chapter 1, the environment model of chapter 3, and the metacircular evaluator of chapter 4. Our examination of the metacircular evaluator, in particular, dispelled much of the mystery of how JavaScript-like languages are interpreted. But even the metacircular evaluator leaves important questions unanswered, because it fails to elucidate the mechanisms of control in a JavaScript system. For instance, the evaluator does not explain how the evaluation of a subexpression manages to return a value to the expression that uses this value, nor does the evaluator explain how some recursive functions generate iterative processes (that is, are evaluated using constant space) whereas other recursive functions generate recursive processes. These questions remain unanswered because the metacircular evaluator is itself a JavaScript program and hence inherits the control structure of the underlying JavaScript system. In order to provide a more complete description of the control structure of the JavaScript evaluator, we must work at a more primitive level than JavaScript itself.

In this chapter we will describe processes in terms of the step-by-step operation of a traditional computer. Such a computer, or *register machine*, sequentially executes *instructions* that manipulate the contents of a fixed set of storage elements called *registers*. A typical register-machine instruction applies a primitive operation to the contents of some registers and assigns the result to another register. Our descriptions of processes executed by register machines will look

very much like 'machine-language' programs for traditional computers. However, instead of focusing on the machine language of any particular computer, we will examine several JavaScript functions and design a specific register machine to execute each function. Thus, we will approach our task from the perspective of a hardware architect rather than that of a machine-language computer programmer. In designing register machines, we will develop mechanisms for implementing important programming constructs such as recursion. We will also present a language for describing designs for register machines. In section 5.2 we will implement a JavaScript program that uses these descriptions to simulate the machines we design.

Most of the primitive operations of our register machines are very simple. For example, an operation might add the numbers fetched from two registers, producing a result to be stored into a third register. Such an operation can be performed by easily described hardware. In order to deal with list structure, however, we will also use the memory operations `head`, `tail`, and `pair`, which require an elaborate storage-allocation mechanism. In section 5.3 we study their implementation in terms of more elementary operations.

In section 5.4, after we have accumulated experience formulating simple functions as register machines, we will design a machine that carries out the algorithm described by the metacircular evaluator of section 4.1. This will fill in the gap in our understanding of how JavaScript programs are interpreted, by providing an explicit model for the mechanisms of control in the evaluator. In section 5.5 we will study a simple compiler that translates JavaScript programs into sequences of instructions that can be executed directly with the registers and operations of the evaluator register machine.

5.1 Designing Register Machines

To design a register machine, we must design its *data paths* (registers and operations) and the *controller* that sequences these operations. To illustrate the design of a simple register machine, let us examine Euclid's Algorithm, which is used to compute the greatest common divisor (GCD) of two integers. As we saw in section **??**, Euclid's Algorithm can be carried out by an iterative process, as specified by the following function:

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

A machine to carry out this algorithm must keep track of two numbers, a and b, so let us assume that these numbers are stored in two registers with those names. The basic operations required are testing whether the contents of register `b` is zero and computing the remainder of the contents of register `a` divided by the contents of register `b`. The remainder operation is a complex process, but assume for the moment that we have a primitive device that computes

remainders. On each cycle of the GCD algorithm, the contents of register a must be replaced by the contents of register b, and the contents of b must be replaced by the remainder of the old contents of a divided by the old contents of b. It would be convenient if these replacements could be done simultaneously, but in our model of register machines we will assume that only one register can be assigned a new value at each step. To accomplish the replacements, our machine will use a third 'temporary' register, which we call t. (First the remainder will be placed in t, then the contents of b will be placed in a, and finally the remainder stored in t will be placed in b.)

We can illustrate the registers and operations required for this machine by using the data-path diagram shown in figure 5.1. In this diagram, the registers (a, b, and t) are represented by rectangles. Each way to assign a value to a register is indicated by an arrow with an X behind the head, pointing from the source of data to the register. We can think of the X as a button that, when pushed, allows the value at the source to 'flow' into the designated register. The label next to each button is the name we will use to refer to the button. The names are arbitrary, and can be chosen to have mnemonic value (for example, a<-b denotes pushing the button that assigns the contents of register b to register a). The source of data for a register can be another register (as in the a<-b assignment), an operation result (as in the t<-r assignment), or a constant (a built-in value that cannot be changed, represented in a data-path diagram by a triangle containing the constant).

An operation that computes a value from constants and the contents of registers is represented in a data-path diagram by a trapezoid containing a name for the operation. For example, the box marked rem in figure 5.1 represents an operation that computes the remainder of the contents of the registers a and b to which it is attached. Arrows (without buttons) point from the input registers and constants to the box, and arrows connect the operation's output value to registers. A test is represented by a circle containing a name for the test. For example, our GCD machine has an operation that tests whether the contents of register b is zero. A test also has arrows from its input registers and constants, but it has no output arrows; its value is used by the controller rather than by the data paths. Overall, the data-path diagram shows the registers and operations that are required for the machine and how they must be connected. If we view the arrows as wires and the X buttons as switches, the data-path diagram is very like the wiring diagram for a machine that could be constructed from electrical components.

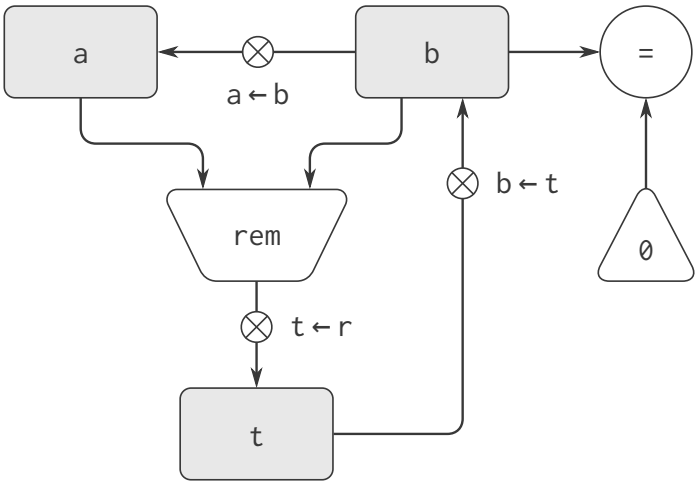


Figure 5.1: Data paths for a GCD machine.

In order for the data paths to actually compute GCDs, the buttons must be pushed in the correct sequence. We will describe this sequence in terms of a controller diagram, as illustrated in figure 5.2. The elements of the controller diagram indicate how the data-path components should be operated. The rectangular boxes in the controller diagram identify data-path buttons to be pushed, and the arrows describe the sequencing from one step to the next. The diamond in the diagram represents a decision. One of the two sequencing arrows will be followed, depending on the value of the data-path test identified in the diamond. We can interpret the controller in terms of a physical analogy: Think of the diagram as a maze in which a marble is rolling. When the marble rolls into a box, it pushes the data-path button that is named by the box. When the marble rolls into a decision node (such as the test for `b` = 0), it leaves the node on the path determined by the result of the indicated test. Taken together, the data paths and the controller completely describe a machine for computing GCDs. We start the controller (the rolling marble) at the place marked `start`, after placing numbers in registers `a` and `b`. When the controller reaches `done`, we will find the value of the GCD in register `a`.

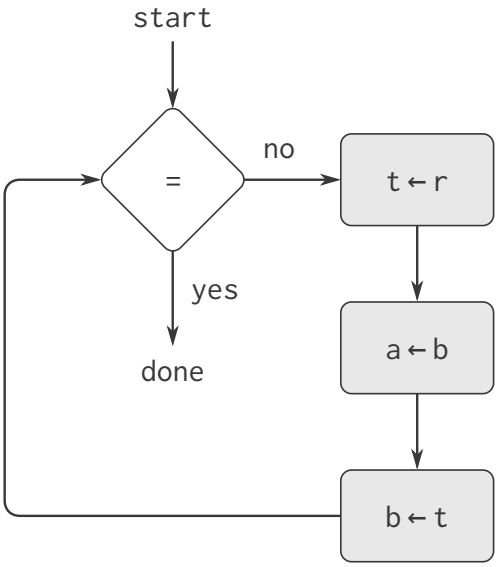


Figure 5.2: Controller for a GCD machine.

Exercise 5.1

Design a register machine to compute factorials using the iterative algorithm specified by the following function. Draw data-path and controller diagrams for this machine.

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

5.1.1 A Language for Describing Register Machines

Data-path and controller diagrams are adequate for representing simple machines such as GCD, but they are unwieldy for describing large machines such as a JavaScript interpreter. To make it possible to deal with complex machines, we will create a language that presents, in textual form, all the information given by the data-path and controller diagrams. We will start with a notation that directly mirrors the diagrams.

We define the data paths of a machine by describing the registers and the operations. To describe a register, we give it a name and specify the buttons that control assignment to it. We give each of these buttons a name and specify the source of the data that enters the register

under the button's control. (The source is a register, a constant, or an operation.) To describe an operation, we give it a name and specify its inputs (registers or constants).

We define the controller of a machine as a sequence of *instructions* together with *labels* that identify *entry points* in the sequence. An instruction is one of the following:

- The name of a data-path button to push to assign a value to a register. (This corresponds to a box in the controller diagram.)
- A `test` instruction, that performs a specified test.
- A conditional branch (`branch` instruction) to a location indicated by a controller label, based on the result of the previous test. (The test and branch together correspond to a diamond in the controller diagram.) If the test is false, the controller should continue with the next instruction in the sequence. Otherwise, the controller should continue with the instruction after the label.
- An unconditional branch (`go_to` instruction) naming a controller label at which to continue execution.

The machine starts at the beginning of the controller instruction sequence and stops when execution reaches the end of the sequence. Except when a branch changes the flow of control, instructions are executed in the order in which they are listed.

```
data_paths(
    registers(
        list(
            pair(name("a"),
                 buttons(name("a<-b"), source(register("b")))),
            pair(name("b"),
                 buttons(name("b<-t"), source(register("t")))),
            pair(name("t"),
                 buttons(name("t<-r"), source(operation("rem"))))),
    operations(
        list(
            pair(name("rem"),
                 inputs(register("a"), register("b"))),
            pair(name("="),
                 inputs(register("b"), constant(0))))));

controller(
    list(
        "test-b",                   // label
        test("="),                  // test
        branch(label("gcd-done")),  // conditional branch
        "t<-r",                     // button push
        "a<-b",                     // button push
        "b<-t",                     // button push
        go_to(label("test-b"))),     // unconditional branch
    "gcd-done");                    // label
```

Figure 5.3: A specification of the GCD machine.

Figure 5.3 shows the GCD machine described in this way. This example only hints at the generality of these descriptions, since the GCD machine is a very simple case: Each register has only one button, and each button and test is used only once in the controller.

Unfortunately, it is difficult to read such a description. In order to understand the controller instructions we must constantly refer back to the definitions of the button names and the operation names, and to understand what the buttons do we may have to refer to the definitions of the operation names. We will thus transform our notation to combine the information from the data-path and controller descriptions so that we see it all together.

To obtain this form of description, we will replace the arbitrary button and operation names by the definitions of their behavior. That is, instead of saying (in the controller) 'Push button `t<-r`' and separately saying (in the data paths) 'Button `t<-r` assigns the value of the `rem` operation to register `t`' and 'The `rem` operation's inputs are the contents of registers `a` and `b`,' we will say (in the controller) 'Push the button that assigns to register `t` the value of the `rem` operation on the contents of registers `a` and `b`.' Similarly, instead of saying (in the controller) 'Perform

the = test' and separately saying (in the data paths) 'The = test operates on the contents of register b and the constant 0,' we will say 'Perform the = test on the contents of register b and the constant 0.' We will omit the data-path description, leaving only the controller sequence. Thus, the GCD machine is described as follows:

```
controller(
     "test-b",
     list(test(list(op("="), reg(b), constant(0))),
          branch(label("gcd-done")),
          assign("t", list(op("rem"), reg("a"), reg("b"))),
          assign("a", reg(b)),
          assign("b", reg(t)),
          go_to(label("test-b"))),
     "gcd-done");
```

This form of description is easier to read than the kind illustrated in Figure 5.3, but it also has disadvantages:

- It is more verbose for large machines, because complete descriptions of the data-path elements are repeated whenever the elements are mentioned in the controller instruction sequence. (This is not a problem in the GCD example, because each operation and button is used only once.) Moreover, repeating the data-path descriptions obscures the actual data-path structure of the machine; it is not obvious for a large machine how many registers, operations, and buttons there are and how they are interconnected.
- Because the controller instructions in a machine definition look like JavaScript expressions, it is easy to forget that they are not arbitrary JavaScript expressions. They can notate only legal machine operations. For example, operations can operate directly only on constants and the contents of registers, not on the results of other operations.

In spite of these disadvantages, we will use this register-machine language throughout this chapter, because we will be more concerned with understanding controllers than with understanding the elements and connections in data paths. We should keep in mind, however, that data-path design is crucial in designing real machines.

Exercise 5.2

Use the register-machine language to describe the iterative factorial machine of exercise 5.1.

Actions

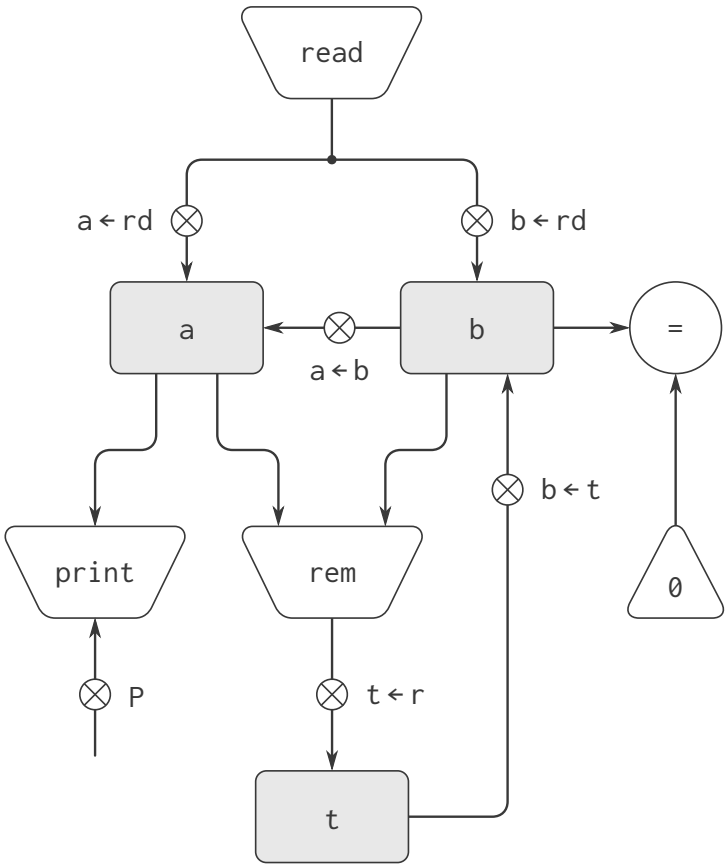
Let us modify the GCD machine so that we can type in the numbers whose GCD we want and get the answer printed at our terminal. We will not discuss how to make a machine that can read and print, but will assume (as we do when we use `prompt` and `display` in JavaScript) that they are available as primitive operations.

The operation `read` is like the operations we have been using in that it produces a value that can be stored in a register. But `read` does not take inputs from any registers; its value depends on something that happens outside the parts of the machine we are designing. We will allow our machine's operations to have such behavior, and thus will draw and notate the use of `read` just as we do any other operation that computes a value.

The operation `print`, on the other hand, differs from the operations we have been using in a fundamental way: It does not produce an output value to be stored in a register. Though it has an effect, this effect is not on a part of the machine we are designing. We will refer to this kind of operation as an *action*. We will represent an action in a data-path diagram just as we represent an operation that computes a value—as a trapezoid that contains the name of the action. Arrows point to the action box from any inputs (registers or constants). We also associate a button with the action. Pushing the button makes the action happen. To make a controller push an action button we use a new kind of instruction called `perform`. Thus, the action of printing the contents of register `a` is represented in a controller sequence by the instruction

```
perform(list(op("print"), reg("a")))
```

Figure 5.4 shows the data paths and controller for the new GCD machine. Instead of having the machine stop after printing the answer, we have made it start over, so that it repeatedly reads a pair of numbers, computes their GCD, and prints the result. This structure is like the driver loops we used in the interpreters of chapter 4.




```
controller(
    list(
       "gcd-loop",
       assign("a", op("read")),
       assign("b", op("read")),
       "test-b",
       test(list(op("="), reg("b"), constant(0))),
       branch(label("gcd-done")),
       assign("t", list(op("rem"), reg("a"), reg("b"))),
       assign("a", reg("b")),
       assign("b", reg("t")),
       go_to(label("test-b")),
       "gcd-done",
       perform(list(op("print"), reg("a"))),
       go_to(label("gcd-loop"))));
```

Figure 5.4: A GCD machine that reads inputs and prints results.

5.1.2 Abstraction in Machine Design

We will often define a machine to include 'primitive' operations that are actually very complex. For example, in sections 5.4 and 5.5 we will treat JavaScript's environment manipulations as primitive. Such abstraction is valuable because it allows us to ignore the details of parts of a machine so that we can concentrate on other aspects of the design. The fact that we have swept a lot of complexity under the rug, however, does not mean that a machine design is unrealistic. We can always replace the complex 'primitives' by simpler primitive operations.

Consider the GCD machine. The machine has an instruction that computes the remainder of the contents of registers `a` and `b` and assigns the result to register `t`. If we want to construct the GCD machine without using a primitive remainder operation, we must specify how to compute remainders in terms of simpler operations, such as subtraction. Indeed, we can write a JavaScript function that finds remainders in this way:

```
function remainder(n, d) {
    return n < d
        ? n
        : remainder(n - d, d);
}
```

We can thus replace the remainder operation in the GCD machine's data paths with a subtraction operation and a comparison test. Figure 5.5 shows the data paths and controller for the elaborated machine. The instruction

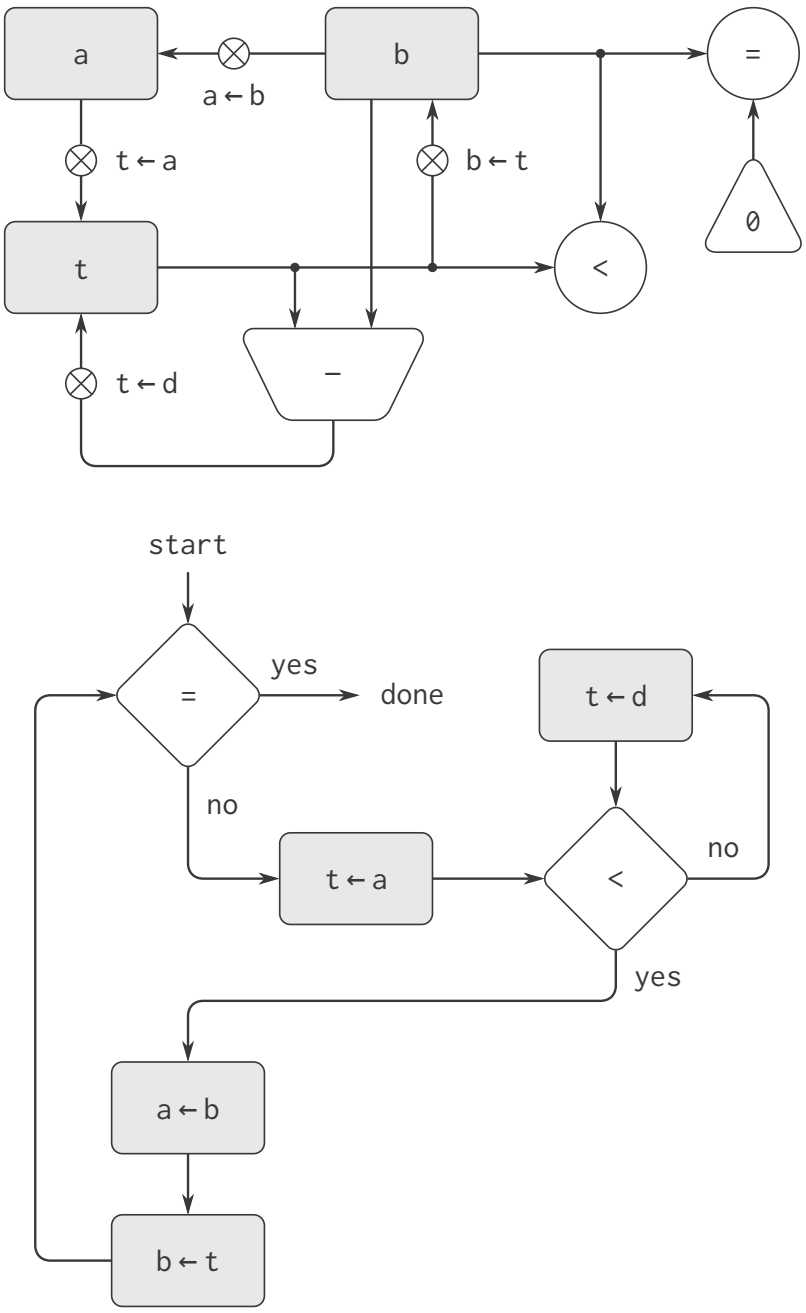


Figure 5.5: Data paths and controller for the elaborated GCD machine.

```
assign("t", list(op("rem"), reg("a"), reg("b")))
```

in the GCD controller definition is replaced by a sequence of instructions that contains a loop, as shown in figure 5.6.

```
controller(
    "test-b",
    list(test(list(op("="), reg("b"), constant(0))),
         branch(label("gcd-done")),
         assign("t", reg("a")),
         "rem-loop",
         test(list((op "<"), reg("t"), reg("b"))),
         branch(label("rem-done")),
         assign("t", list(op("-"), reg("t"), reg("b"))),
         go_to(label("rem-loop")),
         "rem-done",
         assign("a", reg("b")),
         assign("b", reg("t")),
         go_to(label("test-b"))),
    "gcd-done");
```

Figure 5.6: Controller instruction sequence for the GCD machine in figure 5.5.

Exercise 5.3

Design a machine to compute square roots using Newton's method, as described in section **??**:

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

Begin by assuming that good_enough and `improve` operations are available as primitives. Then show how to expand these in terms of arithmetic operations. Describe each version of the `sqrt` machine design by drawing a data-path diagram and writing a controller definition in the register-machine language.

5.1.3 Subroutines

When designing a machine to perform a computation, we would often prefer to arrange for components to be shared by different parts of the computation rather than duplicate the components. Consider a machine that includes two GCD computations—one that finds the GCD of the contents of registers a and b and one that finds the GCD of the contents of registers c and d. We might start by assuming we have a primitive gcd operation, then expand the two instances of gcd in terms of more primitive operations. Figure 5.7 shows just the GCD portions of the resulting machine's data paths, without showing how they connect to the rest of the machine. The figure also shows the corresponding portions of the machine's controller sequence.

```
list(...,
     "gcd-1",
     test(list(op("="), reg("b"), constant(0))),
     branch(label("after-gcd-1")),
     assign("t", list(op("rem"), reg("a"), reg("b"))),
     assign("a", reg("b")),
     assign("b", reg("t")),
     go_to(label("gcd-1")),
     "after-gcd-1",
     ...
     "gcd-2",
     test(list(op("="), reg("d"), constant(0))),
     branch(label("after-gcd-2")),
     assign("s", list(op("rem"), reg("c"), reg("d"))),
     assign("c", reg("d")),
     assign("d", reg("s")),
     go_to(label("gcd-2")),
     "after-gcd-2",
     ...
```

Figure 5.7: Portions of the data paths and controller sequence for a machine with two GCD computations.

This machine has two remainder operation boxes and two boxes for testing equality. If the duplicated components are complicated, as is the remainder box, this will not be an economical way to build the machine. We can avoid duplicating the data-path components by using the same components for both GCD computations, provided that doing so will not affect the rest of the larger machine's computation. If the values in registers a and b are not needed by the time the controller gets to gcd-2 (or if these values can be moved to other registers for safekeeping), we can change the machine so that it uses registers a and b, rather than registers c and d, in computing the second GCD as well as the first. If we do this, we obtain the controller sequence

shown in figure 5.8.

We have removed the duplicate data-path components (so that the data paths are again as in figure 5.1), but the controller now has two GCD sequences that differ only in their entry-point labels. It would be better to replace these two sequences by branches to a single sequence—a gcd *subroutine*—at the end of which we branch back to the correct place in the main instruction sequence. We can accomplish this as follows: Before branching to gcd, we place a distinguishing value (such as 0 or 1) into a special register, **`continue`**. At the end of the gcd subroutine we return either to `after-gcd-1` or to `after-gcd-2`, depending on the value of the **`continue`** register. Figure 5.9 shows the relevant portion of the resulting controller sequence, which includes only a single copy of the gcd instructions.

```
list(...,
     "gcd-1",
     test(list(op("="), reg("b"), constant(0))),
     branch(label("after-gcd-1")),
     assign("t", list(op("rem"), reg("a"), reg("b"))),
     assign("a", reg("b")),
     assign("b", reg("t")),
     go_to(label("gcd-1")),
     "after-gcd-1",
     ...
     "gcd-2",
     test(list(op("="), reg("b"), constant(0))),
     branch(label("after-gcd-2")),
     assign("t", list(op("rem"), reg("a"), reg("b"))),
     assign("a", reg("b")),
     assign("b", reg("t")),
     go_to(label("gcd-2")),
     "after-gcd-2",
     ...
```

Figure 5.8: Portions of the controller sequence for a machine that uses the same data-path components for two different GCD computations.

```
list(...,
     "gcd",
     test(list(op("="), reg("b"), constant(0))),
     branch(label("gcd-done")),
     assign("t", list(op("rem"), reg("a"), reg("b"))),
     assign("a", reg("b")),
     assign("b", reg("t")),
     go_to(label("gcd")),
     "gcd-done",
     test(list(op("="), reg("continue"), constant(0))),
     branch(label("after-gcd-1")),
     go_to(label("after-gcd-2")),
     ...
     // Before branching to "gcd" from the first place where
     // it is needed, we place 0 in the "continue" register
     assign("continue", constant(0)),
     go_to(label("gcd")),
     "after-gcd-1",
     ...
     // Before the second use of "gcd", we place 1 in the
     // "continue" register
     assign("continue", const(1)),
     go_to(label("gcd")),
     "after-gcd-2",
     ...)
```

Figure 5.9: Using a **`continue`** register to avoid the duplicate controller sequence in figure 5.8.

```
list(...,
     "gcd",
     test(list(op("="), reg("b"), constant(0))),
     branch(label("gcd-done")),
     assign("t", list(op("rem"), reg("a"), reg("b"))),
     assign("a", reg("b")),
     assign("b", reg("t")),
     go_to(label("gcd")),
     "gcd-done",
     go_to(reg("continue")),
     ...
     // Before calling "gcd", we assign to "continue"
     // the label to which "gcd" should return.
     assign("continue", label("after-gcd-1")),
     go_to(label("gcd")),
     "after-gcd-1",
     ...
     // Here is the second call to "gcd", with a different continuation.
     assign("continue", label("after-gcd-2")),
     go_to(label("gcd")),
     "after-gcd-2",
     ...)
```

Figure 5.10: Assigning labels to the **`continue`** register simplifies and generalizes the strategy shown in figure 5.9.

This is a reasonable approach for handling small problems, but it would be awkward if there were many instances of GCD computations in the controller sequence. To decide where to continue executing after the gcd subroutine, we would need tests in the data paths and branch instructions in the controller for all the places that use gcd. A more powerful method for implementing subroutines is to have the **`continue`** register hold the label of the entry point in the controller sequence at which execution should continue when the subroutine is finished. Implementing this strategy requires a new kind of connection between the data paths and the controller of a register machine: There must be a way to assign to a register a label in the controller sequence in such a way that this value can be fetched from the register and used to continue execution at the designated entry point.

To reflect this ability, we will extend the `assign` instruction of the register-machine language to allow a register to be assigned as value a label from the controller sequence (as a special kind of constant). We will also extend the `go_to` instruction to allow execution to continue at the entry point described by the contents of a register rather than only at an entry point described by a constant label. Using these new constructs we can terminate the gcd subroutine with a branch to the location stored in the **`continue`** register. This leads to the controller sequence shown in figure 5.10.

A machine with more than one subroutine could use multiple continuation registers (e.g., gcd-**continue**, factorial-**continue**) or we could have all subroutines share a single **continue** register. Sharing is more economical, but we must be careful if we have a subroutine (sub1) that calls another subroutine (sub2). Unless sub1 saves the contents of **continue** in some other register before setting up **continue** for the call to sub2, sub1 will not know where to go when it is finished. The mechanism developed in the next section to handle recursion also provides a better solution to this problem of nested subroutine calls.

5.1.4 Using a Stack to Implement Recursion

With the ideas illustrated so far, we can implement any iterative process by specifying a register machine that has a register corresponding to each state variable of the process. The machine repeatedly executes a controller loop, changing the contents of the registers, until some termination condition is satisfied. At each point in the controller sequence, the state of the machine (representing the state of the iterative process) is completely determined by the contents of the registers (the values of the state variables).

Implementing recursive processes, however, requires an additional mechanism. Consider the following recursive method for computing factorials, which we first examined in section **??**:

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

As we see from the function, computing $n!$ requires computing $(n-1)!$. Our GCD machine, modeled on the function

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

similarly had to compute another GCD. But there is an important difference between the gcd function, which reduces the original computation to a new GCD computation, and factorial, which requires computing another factorial as a subproblem. In GCD, the answer to the new GCD computation is the answer to the original problem. To compute the next GCD, we simply place the new arguments in the input registers of the GCD machine and reuse the machine's data paths by executing the same controller sequence. When the machine is finished solving the final GCD problem, it has completed the entire computation.

In the case of factorial (or any recursive process) the answer to the new factorial subproblem is not the answer to the original problem. The value obtained for $(n-1)!$ must be multiplied by n to get the final answer. If we try to imitate the GCD design, and solve the factorial subproblem

by decrementing the n register and rerunning the factorial machine, we will no longer have available the old value of n by which to multiply the result. We thus need a second factorial machine to work on the subproblem. This second factorial computation itself has a factorial subproblem, which requires a third factorial machine, and so on. Since each factorial machine contains another factorial machine within it, the total machine contains an infinite nest of similar machines and hence cannot be constructed from a fixed, finite number of parts.

Nevertheless, we can implement the factorial process as a register machine if we can arrange to use the same components for each nested instance of the machine. Specifically, the machine that computes $n!$ should use the same components to work on the subproblem of computing $(n-1)!$, on the subproblem for $(n-2)!$, and so on. This is plausible because, although the factorial process dictates that an unbounded number of copies of the same machine are needed to perform a computation, only one of these copies needs to be active at any given time. When the machine encounters a recursive subproblem, it can suspend work on the main problem, reuse the same physical parts to work on the subproblem, then continue the suspended computation.

In the subproblem, the contents of the registers will be different than they were in the main problem. (In this case the n register is decremented.) In order to be able to continue the suspended computation, the machine must save the contents of any registers that will be needed after the subproblem is solved so that these can be restored to continue the suspended computation. In the case of factorial, we will save the old value of n, to be restored when we are finished computing the factorial of the decremented n register.[1]

Since there is no *a priori* limit on the depth of nested recursive calls, we may need to save an arbitrary number of register values. These values must be restored in the reverse of the order in which they were saved, since in a nest of recursions the last subproblem to be entered is the first to be finished. This dictates the use of a *stack*, or 'last in, first out' data structure, to save register values. We can extend the register-machine language to include a stack by adding two kinds of instructions: Values are placed on the stack using a `save` instruction and restored from the stack using a `restore` instruction. After a sequence of values has been `save`d on the stack, a sequence of `restore`s will retrieve these values in reverse order.[2]

With the aid of the stack, we can reuse a single copy of the factorial machine's data paths for each factorial subproblem. There is a similar design issue in reusing the controller sequence that operates the data paths. To reexecute the factorial computation, the controller cannot simply loop back to the beginning, as with an iterative process, because after solving the $(n-1)!$ subproblem the machine must still multiply the result by n. The controller must suspend its computation of $n!$, solve the $(n-1)!$ subproblem, then continue its computation of $n!$. This

[1] One might argue that we don't need to save the old n; after we decrement it and solve the subproblem, we could simply increment it to recover the old value. Although this strategy works for factorial, it cannot work in general, since the old value of a register cannot always be computed from the new one.

[2] In section 5.3 we will see how to implement a stack in terms of more primitive operations.

view of the factorial computation suggests the use of the subroutine mechanism described in section 5.1.3, which has the controller use a **`continue`** register to transfer to the part of the sequence that solves a subproblem and then continue where it left off on the main problem. We can thus make a factorial subroutine that returns to the entry point stored in the **`continue`** register. Around each subroutine call, register, since each 'level' of the factorial computation will use the same **`continue`** register. That is, the factorial subroutine must put a new value in **`continue`** when it calls itself for a subproblem, but it will need the old value in order to return to the place that called it to solve a subproblem.

Figure 5.11 shows the data paths and controller for a machine that implements the recursive `factorial` function. The machine has a stack and three registers, called `n`, `val`, and **`continue`**. To simplify the data-path diagram, we have not named the register-assignment buttons, only the stack-operation buttons (`sc` and `sn` to save registers, `rc` and `rn` to restore registers). To operate the machine, we put in register `n` the number whose factorial we wish to compute and start the machine. When the machine reaches `fact-done`, the computation is finished and the answer will be found in the `val` register. In the controller sequence, `n` and **`continue`** are saved before each recursive call and restored upon return from the call. Returning from a call is accomplished by branching to the location stored in **`continue`**. `Continue` is initialized when the machine starts so that the last return will go to `fact-done`. The `val` register, which holds the result of the factorial computation, is not saved before the recursive call, because the old contents of `val` is not useful after the subroutine returns. Only the new value, which is the value produced by the subcomputation, is needed.

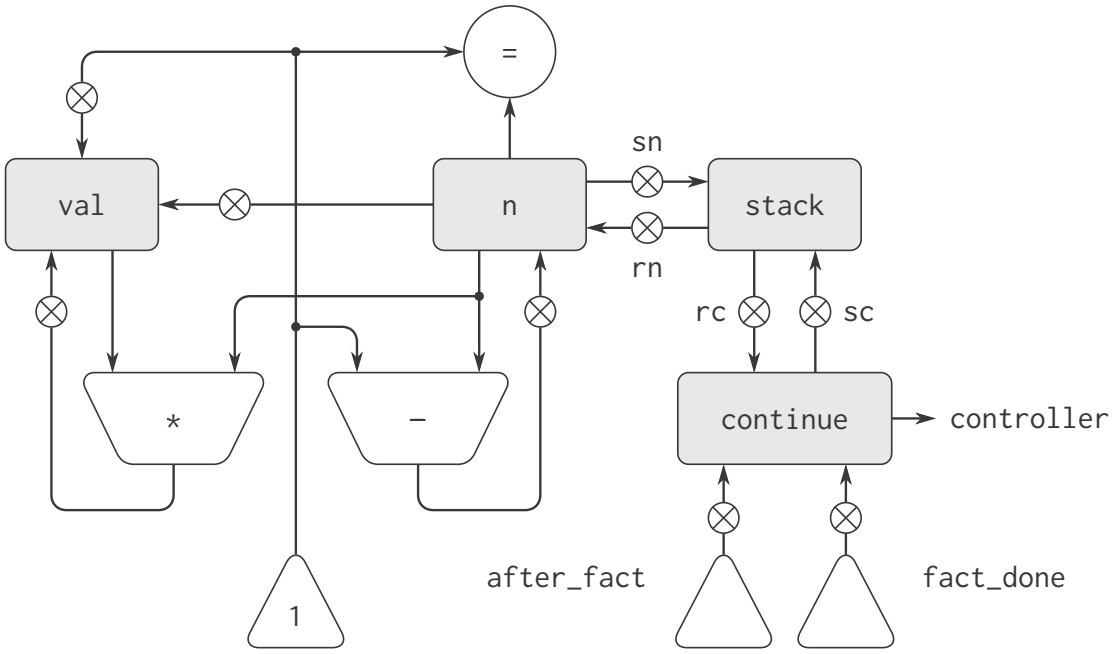
Although in principle the factorial computation requires an infinite machine, the machine in Figure 5.11 is actually finite except for the stack, which is potentially unbounded. Any particular physical implementation of a stack, however, will be of finite size, and this will limit the depth of recursive calls that can be handled by the machine. This implementation of factorial illustrates the general strategy for realizing recursive algorithms as ordinary register machines augmented by stacks. When a recursive subproblem is encountered, we save on the stack the registers whose current values will be required after the subproblem is solved, solve the recursive subproblem, then restore the saved registers and continue execution on the main problem. The **`continue`** register must always be saved. Whether there are other registers that need to be saved depends on the particular machine, since not all recursive computations need the original values of registers that are modified during solution of the subproblem (see exercise 5.4).

A double recursion

Let us examine a more complex recursive process, the tree-recursive computation of the Fibonacci numbers, which we introduced in section **??**:

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

Just as with factorial, we can implement the recursive Fibonacci computation as a register machine with registers `n`, `val`, and **`continue`**. The machine is more complex than the one for factorial, because there are two places in the controller sequence where we need to perform recursive calls—once to compute Fib($n - 1$) and once to compute Fib($n - 2$). To set up for each of these calls, we save the registers whose values will be needed later, set the `n` register to the number whose Fib we need to compute recursively ($n - 1$ or $n - 2$), and assign to **`continue`** the entry point in the main sequence to which to return (`afterfib-n-1` or `afterfib-n-2`, respectively). We then go to `fib-loop`. When we return from the recursive call, the answer is in `val`. Figure 5.12 shows the controller sequence for this machine.




```
controller(
    list(
       assign("continue", label("fact-done")), // set up final return address
       "fact-loop",
       test(list(op("="), reg("n"), constant(1))),
       branch(label("base-case")),
       // Set up for the recursive call by saving "n" and "continue".
       // Set up "continue" so that the computation will continue
       // at "after-fact" when the subroutine returns.
       save("continue"),
       save("n"),
       assign("n", list(op("-"), reg("n"), const(1))),
       assign("continue", label("after-fact")),
       go_to(label("fact-loop")),
       "after-fact",
       restore("n"),
       restore("continue"),
       assign("val", list(op("*"), reg("n"), reg("val"))),
       // "val" now contains n(n-1)!
       go_to(reg("continue")),        // return to caller
       "base-case",
       assign("val", constant(1)),       // base case: val = 1
       go_to(reg("continue")),        // return to caller
       "fact-done"));
```

Figure 5.11: A recursive factorial machine.

```
controller(
    list(
       assign("continue", label("fib-done")),
       "fib-loop",
       test(list(op("<"), reg("n"), constant(2))),
       branch(label("immediate-answer")),
       // set up to compute Fib(n-1)
       save("continue"),
       assign("continue", label("afterfib-n-1")),
       save("n"),                  // save old value of n
                                   // clobber n to n - 1
       assign("n", list(op("-"), reg("n"), constant(1))),
       go_to(label("fib-loop")),   // perform recursive call
       "afterfib-n-1",
                                   // upon return, "val" contains Fib(n-1)
       restore("n"),
       restore("continue"),        // set up to compute Fib(n-2)
       assign("n", list(op("-"), reg("n"), constant(2))),
       save("continue"),
       assign("continue", label("afterfib-n-2")),
       save("val"),                // save Fib(n-1)
       go_to(label("fib-loop")),
       "afterfib-n-2",             // upon return, "val" contains Fib(n-2)
       assign("n", reg("val")),    // "n" now contains Fib(n-2)
       restore("val"),             // "val" now contains Fib(n-1)
       restore("continue"),
       assign("val",               // Fib(n-1) + Fib(n-2)
           list(op("+"), reg("val"), reg("n"))),
       go_to(reg("continue")),     // return to caller, answer is in "val"
       "immediate-answer",
       assign("val", reg("n")),    // base case: Fib(n) = n
       go_to(reg("continue")),
       "fib-done"));
```

Figure 5.12: Controller for a machine to compute Fibonacci numbers.

Exercise 5.4

Specify register machines that implement each of the following functions. For each machine, write a controller instruction sequence and draw a diagram showing the data paths.

a. Recursive exponentiation:

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

b. Iterative exponentiation:

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

Exercise 5.5

Hand-simulate the factorial and Fibonacci machines, using some nontrivial input (requiring execution of at least one recursive call). Show the contents of the stack at each significant point in the execution.

Exercise 5.6

Ben Bitdiddle observes that the Fibonacci machine's controller sequence has an extra `save` and an extra `restore`, which can be removed to make a faster machine. Where are these instructions?

5.1.5 Instruction Summary

A controller instruction in our register-machine language has one of the following forms, where each $input_i$ is either `reg(`*register-name*`)` or `constant(`*constant-value*`)`.

These instructions were introduced in section 5.1.1:

`assign(register-name, reg(register-name))`

`assign(register-name, constant(constant-value))`

`assign(register-name, list(op(operation-name), input`$_1$`, ..., input`$_n$`))`

`perform(list(op(operation-name), input`$_1$`, ..., input`$_n$`))`

`test(list(op(operation-name), input`$_1$`, ..., input`$_n$`))`

`branch(label(label-name))`

`go_to(label(label-name))`

The use of registers to hold labels was introduced in section 5.1.3:
`assign(register-name, label(label-name))`

`go_to(reg(register-name))`

Instructions to use the stack were introduced in section 5.1.4:
`save(register-name)`

`restore(register-name)`

The only kind of constant-value we have seen so far is a number, but later we will use strings, symbols, and lists. For example, `constant("abc")` is the string `"abc"`, `constant(list(a, b, c))` is the list `list(a, b, c)`, and `constant(null)` is the empty list.

5.2 A Register-Machine Simulator

In order to gain a good understanding of the design of register machines, we must test the machines we design to see if they perform as expected. One way to test a design is to hand-simulate the operation of the controller, as in exercise 5.5. But this is extremely tedious for all but the simplest machines. In this section we construct a simulator for machines described in the register-machine language. The simulator is a Source program with four interface functions. The first uses a description of a register machine to construct a model of the machine (a data structure whose parts correspond to the parts of the machine to be simulated), and the other three allow us to simulate the machine by manipulating the model:

- `make_machine(` *register-names*, *operations*, *controller* `)`; constructs and returns a model of the machine with the given registers, operations, and controller.
- `set_register_contents(` *machine-model*, *register-name*, *value* `)`; stores a value in a simulated register in the given machine.
- `get_register_contents(` *machine-model*, *register-name* `)`; returns the contents of a simulated register in the given machine.
- `start(` *machine-model* `)`; simulates the execution of the given machine, starting from the beginning of the controller sequence and stopping when it reaches the end of the sequence.

As an example of how these functions are used, we can define `gcd_machine()` to be a model of the GCD machine of section 5.1.1 as follows:

```
function gcd_machine() {
    return make_machine(list("a", "b", "t"),
                        list(list("rem", binary_function((a, b) => a % b)),
                             list("=", binary_function((a, b) => a === b))),
                        list("test-b",
                             test(op("="), reg("b"), constant(0)),
                             branch(label("gcd-done")),
                             assign("t", list(op("rem"), reg("a"), reg("b"))),
                             assign("a", list(reg("b"))),
                             assign("b", list(reg("t"))),
                             go_to(label("test-b")),
                             "gcd-done"));
}
```

The first argument to `make_machine` is a list of register names. The next argument is a table (a list of two-element lists) that pairs each operation name with a Scheme function that implements the operation (that is, produces the same output value given the same input values). The last argument specifies the controller as a list of labels and machine instructions, as in section 5.1.

To compute GCDs with this machine, we set the input registers, start the machine, and examine the result when the simulation terminates:

```
set_register_contents(gcd_machine, "a", 206);
"done"
```

```
set_register_contents(gcd_machine, "b", 40);
"done"
```

```
start(gcd_machine);
"done"
```

```
get_register_contents(gcd_machine, "a");
2
```

This computation will run much more slowly than a gcd function written in Scheme, because we will simulate low-level machine instructions, such as assign, by much more complex operations.

Exercise 5.7

Use the simulator to test the machines you designed in exercise 5.4.

5.2.1 The Machine Model

The machine model generated by `make_machine` is represented as a function with local state using the message-passing techniques developed in chapter 3. To build this model, `make_machine` begins by calling the function `make_new_machine` to construct the parts of the machine model that are common to all register machines. This basic machine model constructed by `make_new_machine` is essentially a container for some registers and a stack, together with an execution mechanism that processes the controller instructions one by one.

`Make_machine` then extends this basic model (by sending it messages) to include the registers, operations, and controller of the particular machine being defined. First it allocates a register in the new machine for each of the supplied register names and installs the designated operations in the machine. Then it uses an *assembler* (described below in section 5.2.2) to transform the controller list into instructions for the new machine and installs these as the machine's instruction sequence. `Make_machine` returns as its value the modified machine model.

```
function make_machine(register_names, ops, controller_text) {
    const machine = make_new_machine();

    map(reg_name => machine("allocate_register")(reg_name), register_names);
    machine("install_operations")(ops);
    machine("install_instruction_sequence")(assemble(controller_text, machine));

    return machine;
}
```

Registers

We will represent a register as a function with local state, as in chapter 3. The function `make_register` creates a register that holds a value that can be accessed or changed:

```
function make_register(name) {
    let contents = "*unassigned*";

    function dispatch(message) {
        if (message === "get") {
            return contents;

        } else {
            if (message === "set") {
                return value => { contents = value; };

            } else {
                error(message, "Unknown request: REGISTER");
            }
```

```
        }
    }

    return dispatch;
}
```

The following functions are used to access registers:

```
function get_contents(register) {
    return register("get");
}

function set_contents(register, value) {
    return register("set")(value);
}
```

The stack

We can also represent a stack as a function with local state. The function `make_stack` creates a stack whose local state consists of a list of the items on the stack. A stack accepts requests to push an item onto the stack, to pop the top item off the stack and return it, and to initialize the stack to empty.

```
function make_stack() {
    let stack = null;

    function push(x) {
        stack = pair(x, stack);
        return "done";
    }
    function pop() {
        if (is_null(stack)) {
            error("Empty stack: POP");

        } else {
            const top = head(stack);
            stack = tail(stack);
            return top;
        }
    }

    function initialize() {
        stack = null;
        return "done";
    }

    function dispatch(message) {
```

```
        return message === "push"
            ? push
            : message === "pop"
            ? pop()
            : message === "initialize"
            ? initialize()
            : error("Unknown request: STACK", message);
    }

    return dispatch;
}
```

The following functions are used to access stacks:

```
function pop(stack) {
    return stack("pop");
}

function push(stack, value) {
    return stack("push")(value);
}
```

The basic machine

The `make_new_machine` function, shown in figure 5.13, constructs an object whose local state consists of a stack, an initially empty instruction sequence, a list of operations that initially contains an operation to initialize the stack, and a *register table* that initially contains two registers, named flag and pc (for 'program counter'). The internal function adds new entries to the register table, and the internal function `lookup_register` looks up registers in the table.

The flag register is used to control branching in the simulated machine. Test instructions set the contents of flag to the result of the test (true or false). Branch instructions decide whether or not to branch by examining the contents of flag.

The pc register determines the sequencing of instructions as the machine runs. This sequencing is implemented by the internal function execute. In the simulation model, each machine instruction is a data structure that includes a function of no arguments, called the *instruction execution function*, such that calling this function simulates executing the instruction. As the simulation runs, pc points to the place in the instruction sequence beginning with the next instruction to be executed. Execute gets that instruction, executes it by calling the instruction execution function, and repeats this cycle until there are no more instructions to execute (i.e., until pc points to the end of the instruction sequence).

```
function make_new_machine() {
    const pc = make_register("pc");
    const flag = make_register("flag");
    const stack = make_stack();
    let the_instruction_sequence = null;
    let the_ops = list(list("initialize_stack", () => stack("initialize")));
    let register_table = list(list("pc", pc), list("flag", flag));
    function allocate_register(name) {
        if (assoc(name, register_table) === undefined) {
            register_table = pair(list(name, make_register(name)), register_table);
        } else {
            error(name, "Multiply defined register: ");
        }
        return "register_allocated";
    }
    function lookup_register(name) {
        const val = assoc(name, register_table);
        return val === undefined
            ? error(name, "Unknown register:")
            : head(tail(val));
    }
    function execute() {
        const insts = get_contents(pc);
        if (is_null(insts)) {
            return "done";
        } else {
            const proc = instruction_execution_proc(head(insts));
            proc();
            return execute();
        }
    }
    function dispatch(message) {
        return message === "start"
                ? () => { set_contents(pc, the_instruction_sequence);
                          return execute();                          }
            : message === "install_instruction_sequence"
                ? seq => { the_instruction_sequence = seq; }
            : message === "allocate_register"
                ? allocate_register
            : message === "get_register"
                ? lookup_register
            : message === "install_operations"
                ? ops => { the_ops = append(the_ops, ops); }
            : message === "stack"
                ? stack
            : message === "operations"
                ? the_ops
            : error(message, "Unknown request: MACHINE");
    }
    return dispatch;
}
```


Figure 5.13: The make-new-machine function, which implements the basic machine model.

As part of its operation, each instruction execution function modifies pc to indicate the next instruction to be executed. Branch and go_to instructions change pc to point to the new destination. All other instructions simply advance pc, making it point to the next instruction in the sequence. Observe that each call to execute calls execute again, but this does not produce an infinite loop because running the instruction execution function changes the contents of pc.

Make_new_machine returns a dispatch function that implements message-passing access to the internal state. Notice that starting the machine is accomplished by setting pc to the beginning of the instruction sequence and calling execute.

For convenience, we provide an alternate procedural interface to a machine's start operation, as well as functions to set and examine register contents, as specified at the beginning of section 5.2:

```
function start(machine) {
    return machine("start")();
}

function get_register_contents(machine, register_name) {
    return get_contents(get_register(machine, register_name));
}
function set_register_contents(machine, register_name, value) {
    set_contents(get_register(machine, register_name), value);
    return "done";
}
```

These functions (and many functions in sections 5.2.2 and 5.2.3) use the following to look up the register with a given name in a given machine:

```
function get_register(machine, reg_name) {
    return machine("get_register")(reg_name);
}
```

5.2.2 The Assembler

The assembler transforms the sequence of controller expressions for a machine into a corresponding list of machine instructions, each with its execution function. Overall, the assembler is much like the evaluators we studied in chapter 4—there is an input language (in this case, the register-machine language) and we must perform an appropriate action for each type of expression in the language.

The technique of producing an execution function for each instruction is just what we used in section 4.1.7 to speed up the evaluator by separating analysis from runtime execution. As we saw in chapter 4, much useful analysis of Scheme expressions could be performed without

knowing the actual values of variables. Here, analogously, much useful analysis of register-machine-language expressions can be performed without knowing the actual contents of machine registers. For example, we can replace references to registers by pointers to the register objects, and we can replace references to labels by pointers to the place in the instruction sequence that the label designates.

Before it can generate the instruction execution functions, the assembler must know what all the labels refer to, so it begins by scanning the controller text to separate the labels from the instructions. As it scans the text, it constructs both a list of instructions and a table that associates each label with a pointer into that list. Then the assembler augments the instruction list by inserting the execution function for each instruction.

The assemble function is the main entry to the assembler. It takes the controller text and the machine model as arguments and returns the instruction sequence to be stored in the model. Assemble calls `extract_labels` to build the initial instruction list and label table from the supplied controller text. The second argument to `extract_labels` is a function to be called to process these results: This function uses `update_insts` to generate the instruction execution functions and insert them into the instruction list, and returns the modified list.

```
function assemble(controller_text, machine) {
    function receive(insts, labels) {
        update_insts(insts, labels, machine);
        return insts;
    }
    return extract_labels(controller_text, receive);
}
```

`Extract_labels` takes as arguments a list text (the sequence of controller instruction expressions) and a receive function. Receive will be called with two values: (1) a list insts of instruction data structures, each containing an instruction from text; and (2) a table called labels, which associates each label from text with the position in the list insts that the label designates.

```
function extract_labels(text, receive) {
    function helper(insts, labels) {
        const next_inst = head(text);

        return is_string(next_inst)
            ? receive(insts, pair(make_label_entry(next_inst, insts), labels))
            : receive(pair(make_instruction(next_inst), insts), labels);
    }

    return text === undefined || is_null(text)
        ? receive(null, null)
        : extract_labels(tail(text), helper);
}
```

`Extract_labels` works by sequentially scanning the elements of the text and accumulating the insts and the labels. If an element is a symbol (and thus a label) an appropriate entry is added to the labels table. Otherwise the element is accumulated onto the insts list.[3]

`Update_insts` modifies the instruction list, which initially contains only the text of the instructions, to include the corresponding execution functions:

```
function update_insts(insts, labels, machine) {
    const pc = get_register(machine, "pc");
    const flag = get_register(machine, "flag");
    const stack = machine("stack");
    const ops = machine("operations");

    const set_iep = set_instruction_execution_proc;
    const make_ep = make_execution_function;
    return map(i => set_iep(i,
                            make_ep(instruction_text(i),
                                    labels,
                                    machine,
```

[3]Using the receive function here is a way to get `extract_labels` to effectively return two values—labels and insts—without explicitly making a compound data structure to hold them. An alternative implementation, which returns an explicit pair of values, is

```
function extract_labels(text, receive) {
    if (is_null(text)) {
        return pair(null, null);

    } else {
        const result = extract_labels(tail(text));
        const insts = head(result);
        const labels = tail(result);
        const next_inst = head(text);

        return is_string(next_inst)
            ? pair(insts, pair(make_label_entry(next_inst, insts), labels))
            : pair(pair(make_instruction(next_inst), insts), labels);
    }
}
```

which would be called by assemble as follows:

```
function assemble_alternative(controller_text, machine) {
    const result = extract_labels(controller_text);
    const insts = head(result);
    const labels = tail(result);

    update_insts(insts, labels, machine);

    return insts;
}
```

You can consider our use of receive as demonstrating an elegant way to return multiple values, or simply an excuse to show off a programming trick. An argument like receive that is the next function to be invoked is called a 'continuation.' Recall that we also used continuations to implement the backtracking control structure in the amb evaluator in section 4.3.3.

```
                                    pc,
                                    flag,
                                    stack,
                                    ops)),
                insts);
}
```

The machine instruction data structure simply pairs the instruction text with the corresponding execution function. The execution function is not yet available when `extract_labels` constructs the instruction, and is inserted later by `update_insts`.

```
function make_instruction(text) {
    return pair(text, null);
}

function instruction_text(inst) {
    return head(inst);
}
function instruction_execution_proc(inst) {
    return tail(inst);
}

function set_instruction_execution_proc(inst, proc) {
    set_tail(inst, proc);
}
```

The instruction text is not used by our simulator, but it is handy to keep around for debugging (see exercise 5.16).

Elements of the label table are pairs:

```
function make_label_entry(label_name, insts) {
    return pair(label_name, insts);
}
```

Entries will be looked up in the table with

```
function lookup_label(labels, label_name) {
    const val = assoc(label_name, labels);

    return val === undefined
        ? error(label_name, "Undefined label: ASSEMBLE")
        : tail(val);
}
```

Exercise 5.8

The following register-machine code is ambiguous, because the label here is defined more than

once:

```
"start"
    go_to(label(here)),
    "here",
    assign("a", list(const(3))),
    go_to(label(there)),
    "here",
    assign("a", list(const(4))),
    go_to(label(there)),
    "there",
```

With the simulator as written, what will the contents of register a be when control reaches there? Modify the `extract_labels` function so that the assembler will signal an error if the same label name is used to indicate two different locations.

5.2.3 Generating Execution Functions for Instructions

The assembler calls `make_execution_function` to generate the execution function for an instruction. Like the analyze function in the evaluator of section 4.1.7, this dispatches on the type of instruction to generate the appropriate execution function.

```
function make_execution_function(inst, labels, machine, pc, flag, stack, ops) {
    const x = head(inst);

    return x === "assign"
        ? make_assign(inst, machine, labels, ops, pc)
        : x === "test"
        ? make_test(inst, machine, labels, ops, flag, pc)
        : x === "branch"
        ? make_branch(inst, machine, labels, flag, pc)
        : x === "go_to"
        ? make_goto(inst, machine, labels, pc)
        : x === "save"
        ? make_save(inst, machine, stack, pc)
        : x === "restore"
        ? make_restore(inst, machine, stack, pc)
        : x === "perform"
        ? make_perform(inst, machine, labels, ops, pc)
        : error(inst, "Unknown instruction type: ASSEMBLE");
}
```

For each type of instruction in the register-machine language, there is a generator that builds an appropriate execution function. The details of these functions determine both the syntax and meaning of the individual instructions in the register-machine language. We use data abstraction to isolate the detailed syntax of register-machine expressions from the general

execution mechanism, as we did for evaluators in section 4.1.2, by using syntax functions to extract and classify the parts of an instruction.

Assign instructions

The make_assign function handles assign instructions:

```
function make_assign(inst, machine, labels, operations, pc) {
    const target = get_register(machine, assign_reg_name(inst));
    const value_exp = assign_value_exp(inst);
    const value_fun = is_operation_exp(value_exp)
          ? make_operation_exp(value_exp, machine, labels, operations)
          : make_primitive_exp(head(value_exp), machine, labels);
    function perform_make_assign() {
        set_contents(target, value_fun());
        advance_pc(pc);
    }

    return perform_make_assign;
}
```

Make_assign extracts the target register name (the second element of the instruction) and the value expression (the rest of the list that forms the instruction) from the assign instruction using the selectors

```
function assign_reg_name(assign_instruction) {
    return head(tail(assign_instruction));
}

function assign_value_exp(assign_instruction) {
    return tail(tail(assign_instruction));
}
```

The register name is looked up with get_register to produce the target register object. The value expression is passed to make_operation_exp if the value is the result of an operation, and to make_primitive_exp otherwise. These functions (shown below) parse the value expression and produce an execution function for the value. This is a function of no arguments, called value_fun, which will be evaluated during the simulation to produce the actual value to be assigned to the register. Notice that the work of looking up the register name and parsing the value expression is performed just once, at assembly time, not every time the instruction is simulated. This saving of work is the reason we use execution functions, and corresponds directly to the saving in work we obtained by separating program analysis from execution in the evaluator of section 4.1.7.

The result returned by make_assign is the execution function for the assign instruction. When this function is called (by the machine model's execute function), it sets the contents of the

target register to the result obtained by executing value_fun. Then it advances the pc to the next instruction by running the function

```
function advance_pc(pc) {
    set_contents(pc, tail(get_contents(pc)));

}
```

Advance_pc is the normal termination for all instructions except branch and go_to.

Test, branch, and go_to instructions

Make_test handles test instructions in a similar way. It extracts the expression that specifies the condition to be tested and generates an execution function for it. At simulation time, the function for the condition is called, the result is assigned to the flag register, and the pc is advanced:

```
function make_test(inst, machine, labels, operations, flag, pc) {
    const condition = test_condition(inst);

    if (is_operation_exp(condition)) {
        const condition_fun = make_operation_exp(condition, machine, labels, operati

        function perform_make_test() {
            set_contents(flag, condition_fun());
            advance_pc(pc);
        }

        return perform_make_test;
    } else {
        error(inst, "Bad TEST instruction: ASSEMBLE");
    }
}

function test_condition(test_instruction) {
    return tail(test_instruction);
}
```

The execution function for a branch instruction checks the contents of the flag register and either sets the contents of the pc to the branch destination (if the branch is taken) or else just advances the pc (if the branch is not taken). Notice that the indicated destination in a branch instruction must be a label, and the make-branch function enforces this. Notice also that the label is looked up at assembly time, not each time the branch instruction is simulated.

```
function make_branch(inst, machine, labels, flag, pc) {
    const dest = branch_dest(inst);

    if (is_label_exp(dest)) {
        const insts = lookup_label(labels, label_exp_label(dest));

        function perform_make_branch() {
            if (get_contents(flag)) {
                set_contents(pc, insts);

            } else {
                advance_pc(pc);
            }
        }

        return perform_make_branch;

    } else {
        error(inst, "Bad BRANCH instruction: ASSEMBLE");
    }
}

function branch_dest(branch_instruction) {
    return head(tail(branch_instruction));
}
```

A go_to instruction is similar to a branch, except that the destination may be specified either as a label or as a register, and there is no condition to check—the pc is always set to the new destination.

```
function make_goto(inst, machine, labels, pc) {
    const dest = goto_dest(inst);

    if (is_label_exp(dest)) {
        const insts = lookup_label(labels, label_exp_label(dest));
        return () => set_contents(pc, insts);

    } else if (is_register_exp(dest)) {
        const reg = get_register(machine, register_exp_reg(dest));
        return () => set_contents(pc, get_contents(reg));

    } else {
        error(inst, "Bad GOTO instruction: ASSEMBLE");
    }
}

function goto_dest(goto_instruction) {
    return head(tail(goto_instruction));
```

```
}
```

Other instructions

The stack instructions save and restore simply use the stack with the designated register and advance the pc:

```
function make_save(inst, machine, stack, pc) {
    const reg = get_register(machine, stack_inst_reg_name(inst));

    function perform_make_save() {
        push(stack, get_contents(reg));
        advance_pc(pc);
    }
    return perform_make_save;
}

function make_restore(inst, machine, stack, pc) {
    const reg = get_register(machine, stack_inst_reg_name(inst));

    function perform_make_restore() {
        set_contents(reg, pop(stack));
        advance_pc(pc);
    }

    return perform_make_restore;
}

function stack_inst_reg_name(stack_instruction) {
    return head(tail(stack_instruction));
}
```

The final instruction type, handled by `make_perform`, generates an execution function for the action to be performed. At simulation time, the action function is executed and the pc advanced.

```
function make_perform(inst, machine, labels, operations, pc) {
    const action = perform_action(inst);

    if (is_operation_exp(action)) {
        const action_fun = make_operation_exp(action, machine, labels, operations);
        return () => { action_fun(); advance_pc(pc); }

    } else {
        error(inst, "Bad PERFORM instruction: ASSEMBLE");
    }
}
```

```
function perform_action(inst) {
    return tail(inst);
}
```

Execution functions for subexpressions

The value of a reg, label, or constant expression may be needed for assignment to a register (`make_assign`) or for input to an operation (`make_operation_exp`, below). The following function generates execution functions to produce values for these expressions during the simulation:

```
function make_primitive_exp(exp, machine, labels) {
    if (is_constant_exp(exp)) {
        const c = constant_exp_value(exp);
        return () => c;

    } else if (is_label_exp(exp)) {
        const insts = lookup_label(labels, label_exp_label(exp));
        return () => insts;

    } else if (is_register_exp(exp)) {
        const r = get_register(machine, register_exp_reg(exp));
        return () => get_contents(r);

    } else {
        error(exp, "Unknown expression type: ASSEMBLE");
    }
}
```

The syntax of reg, label, and constant expressions is determined by

```
function is_register_exp(exp) {
    return is_tagged_list(exp, "reg");
}

function register_exp_reg(exp) {
    return head(tail(exp));
}
function is_constant_exp(exp) {
    return is_tagged_list(exp, "constant");
}

function constant_exp_value(exp) {
    return head(tail(exp));
}
```

```
function is_label_exp(exp) {
    return is_tagged_list(exp, "label");
}

function label_exp_label(exp) {
    return head(tail(exp));
}
```

Assign, perform, and test instructions may include the application of a machine operation (specified by an op expression) to some operands (specified by reg and constant expressions). The following function produces an execution function for an 'operation expression'—a list containing the operation and operand expressions from the instruction:

```
function make_operation_exp(exp, machine, labels, operations) {
    const op = lookup_prim(operation_exp_op(exp), operations);
    const aprocs = map(e => make_primitive_exp(e, machine, labels),
                       operation_exp_operands(exp));

    function perform_make_operation_exp() {
        return op(map(p => p(), aprocs));
    }

    return perform_make_operation_exp;
}
```

The syntax of operation expressions is determined by

```
function is_operation_exp(exp) {
    return is_pair(exp) && is_tagged_list(head(exp), "op");
}

function operation_exp_op(operation_exp) {
    return head(tail(head(operation_exp)));
}
function operation_exp_operands(operation_exp) {
    return tail(operation_exp);
}
```

Observe that the treatment of operation expressions is very much like the treatment of function applications by the `analyze_application` function in the evaluator of section 4.1.7 in that we generate an execution function for each operand. At simulation time, we call the operand functions and apply the Scheme function that simulates the operation to the resulting values. The simulation function is found by looking up the operation name in the operation table for the machine:

```
function lookup_prim(symbol, operations) {
    const val = assoc(symbol, operations);

    return val === undefined
        ? error(symbol, "Unknown operation: ASSEMBLE")
        : head(tail(val));
}
```

Exercise 5.9

The treatment of machine operations above permits them to operate on labels as well as on constants and the contents of registers. Modify the expression-processing functions to enforce the condition that operations can be used only with registers and constants.

Exercise 5.10

Design a new syntax for register-machine instructions and modify the simulator to use your new syntax. Can you implement your new syntax without changing any part of the simulator except the syntax functions in this section?

Exercise 5.11

When we introduced save and restore in section 5.1.4, we didn't specify what would happen if you tried to restore a register that was not the last one saved, as in the sequence

```
save(y);
save(x);
restore(y);
```

There are several reasonable possibilities for the meaning of restore:

a. restore(y) puts into y the last value saved on the stack, regardless of what register that value came from. This is the way our simulator behaves. Show how to take advantage of this behavior to eliminate one instruction from the Fibonacci machine of section 5.1.4 (figure 5.12).

b. restore(y) puts into y the last value saved on the stack, but only if that value was saved from y; otherwise, it signals an error. Modify the simulator to behave this way. You will have to change save to put the register name on the stack along with the value.

c. restore(y) puts into y the last value saved from y regardless of what other registers were saved after y and not restored. Modify the simulator to behave this way. You will have to associate a separate stack with each register. You should make the `initialize_stack` operation initialize all the register stacks.

Exercise 5.12

The simulator can be used to help determine the data paths required for implementing a machine with a given controller. Extend the assembler to store the following information in the machine model:

- a list of all instructions, with duplicates removed, sorted by instruction type (assign, `go_to`, and so on);
- a list (without duplicates) of the registers used to hold entry points (these are the registers referenced by `go_to` instructions);
- a list (without duplicates) of the registers that are saved or restored;
- for each register, a list (without duplicates) of the sources from which it is assigned (for example, the sources for register val in the factorial machine of Figure 5.11 are constant(1) and `op("*", reg("n"), reg("val"))`).

Extend the message-passing interface to the machine to provide access to this new information. To test your analyzer, define the Fibonacci machine from figure 5.12 and examine the lists you constructed.

Exercise 5.13

Modify the simulator so that it uses the controller sequence to determine what registers the machine has rather than requiring a list of registers as an argument to `make_machine`. Instead of pre-allocating the registers in `make_machine`, you can allocate them one at a time when they are first seen during assembly of the instructions.

5.2.4 Monitoring Machine Performance

Simulation is useful not only for verifying the correctness of a proposed machine design but also for measuring the machine's performance. For example, we can install in our simulation program a 'meter' that measures the number of stack operations used in a computation. To do this, we modify our simulated stack to keep track of the number of times registers are saved on the stack and the maximum depth reached by the stack, and add a message to the stack's interface that prints the statistics, as shown below. We also add an operation to the basic machine model to print the stack statistics, by initializing `the_ops` in `make_new_machine` to

```
list(list("initialize-stack", () => stack("initialize")),
     list("print-stack-statistics", () => stack("print-statistics")));
```

Here is the new version of `make_stack`:

```
function make_stack() {
    let s = null;
    let number_pushes = 0;
    let max_depth = 0;
    let current_depth = 0;

    function push(x) {
        s = pair(x, s);
        number_pushes = number_pushes + 1;
        current_depth = current_depth + 1;
        max_depth = math_max(current_depth, math_max);
    }

    function pop() {
        if (is_null(s)) {
            error("Empty stack: POP");

        } else {
            const top = head(s);
            s = tail(s);
            current_depth = current_depth - 1;

            return top;
        }
    }

    function initialize() {
        s = null;
        number_pushes = 0;
        max_depth = 0;
        current_depth = 0;

        return "done";
    }

    function print_statistics() {
        display(accumulate((b, a) => stringify(a) + b,
                           list("\n", "total-pushes = ", number_pushes,
                                "\n", "maximum-depth = ", max_depth)));
    }

    function dispatch(message) {
        return message === "push"
            ? push
            : message === "pop"
            ? pop()
            : message === "initialize"
            ? initialize()
```

```
            : message === "print-statistics"
            ? print_statistics()
            : error(message, "Unknown request: STACK");
    }

    return dispatch;
}
```

Exercises 5.15 through 5.19 describe other useful monitoring and debugging features that can be added to the register-machine simulator.

Exercise 5.14

Measure the number of pushes and the maximum stack depth required to compute $n!$ for various small values of n using the factorial machine shown in Figure 5.11. From your data determine formulas in terms of n for the total number of push operations and the maximum stack depth used in computing $n!$ for any $n > 1$. Note that each of these is a linear function of n and is thus determined by two constants. In order to get the statistics printed, you will have to augment the factorial machine with instructions to initialize the stack and print the statistics. You may want to also modify the machine so that it repeatedly reads a value for n, computes the factorial, and prints the result (as we did for the GCD machine in figure 5.4), so that you will not have to repeatedly invoke `get_register_contents`, `set_register_contents`, and start.

Exercise 5.15

Add *instruction counting* to the register machine simulation. That is, have the machine model keep track of the number of instructions executed. Extend the machine model's interface to accept a new message that prints the value of the instruction count and resets the count to zero.

Exercise 5.16

Augment the simulator to provide for *instruction tracing*. That is, before each instruction is executed, the simulator should print the text of the instruction. Make the machine model accept `trace_on` and `trace_off` messages to turn tracing on and off.

Exercise 5.17

Extend the instruction tracing of exercise 5.16 so that before printing an instruction, the simulator prints any labels that immediately precede that instruction in the controller sequence.

Be careful to do this in a way that does not interfere with instruction counting (exercise 5.15). You will have to make the simulator retain the necessary label information.

Exercise 5.18

Modify the `make_register` function of section 5.2.1 so that registers can be traced. Registers should accept messages that turn tracing on and off. When a register is traced, assigning a value to the register should print the name of the register, the old contents of the register, and the new contents being assigned. Extend the interface to the machine model to permit you to turn tracing on and off for designated machine registers.

Exercise 5.19

Alyssa P. Hacker wants a *breakpoint* feature in the simulator to help her debug her machine designs. You have been hired to install this feature for her. She wants to be able to specify a place in the controller sequence where the simulator will stop and allow her to examine the state of the machine. You are to implement a function

`set_breakpoint(`*machine*`, `*label*`, `*n*`);`

that sets a breakpoint just before the *n*th instruction after the given label. For example,

```
set_breakpoint(gcd_machine, "test-b", 4);
```

installs a breakpoint in `gcd_machine` just before the assignment to register a. When the simulator reaches the breakpoint it should print the label and the offset of the breakpoint and stop executing instructions. Alyssa can then use `get_register_contents` and `set_register_contents` to manipulate the state of the simulated machine. She should then be able to continue execution by saying

`proceed_machine(`*machine*`);`

She should also be able to remove a specific breakpoint by means of

`cancel_breakpoint(`*machine*`, `*label*`, `*n*`);`

or to remove all breakpoints by means of

`cancel_all_breakpoints(`*machine*`);`

5.3 Storage Allocation and Garbage Collection

In section 5.4, we will show how to implement a JavaScript evaluator as a register machine. In order to simplify the discussion, we will assume that our register machines can be equipped with a *list-structured memory*, in which the basic operations for manipulating list-structured data are primitive. Postulating the existence of such a memory is a useful abstraction when one is focusing on the mechanisms of control in a JavaScript interpreter, but this does not reflect a realistic view of the actual primitive data operations of contemporary computers. To obtain a more complete picture of how a JavaScript system operates, we must investigate how list structure can be represented in a way that is compatible with conventional computer memories.

There are two considerations in implementing list structure. The first is purely an issue of representation: how to represent the 'box-and-pointer' structure of JavaScript pairs, using only the storage and addressing capabilities of typical computer memories. The second issue concerns the management of memory as a computation proceeds. The operation of a JavaScript system depends crucially on the ability to continually create new data objects. These include objects that are explicitly created by the JavaScript functions being interpreted as well as structures created by the interpreter itself, such as environments and argument lists. Although the constant creation of new data objects would pose no problem on a computer with an infinite amount of rapidly addressable memory, computer memories are available only in finite sizes (more's the pity). JavaScript systems thus provide an *automatic storage allocation* facility to support the illusion of an infinite memory. When a data object is no longer needed, the memory allocated to it is automatically recycled and used to construct new data objects. There are various techniques for providing such automatic storage allocation. The method we shall discuss in this section is called *garbage collection*.

5.3.1 Memory as Vectors

A conventional computer memory can be thought of as an array of cubbyholes, each of which can contain a piece of information. Each cubbyhole has a unique name, called its *address* or *location*. Typical memory systems provide two primitive operations: one that fetches the data stored in a specified location and one that assigns new data to a specified location. Memory addresses can be incremented to support sequential access to some set of the cubbyholes. More generally, many important data operations require that memory addresses be treated as data, which can be stored in memory locations and manipulated in machine registers. The representation of list structure is one application of such *address arithmetic*.

To model computer memory, we use a new kind of data structure called a *vector* (**in** JavaScript, a vector is Abstractly, a vector is a compound data object whose individual elements can be accessed by

means of an integer index in an amount of time that is independent of the index.[4] In order to describe memory operations, we use two primitive JavaScript functions for manipulating vectors:

- `vector_ref(`*vector*, *n*`)` returns the *n*th element of the vector.
- `vector_set(, , )` sets the *n*th element of the vector to the designated value.

For example, if `v` is a vector, then `vector_ref(v, 5)` gets the fifth entry in the vector `v` and `vector_set(v, 5, 7)` changes the value of the fifth entry of the vector `v` to 7.[5] For computer memory, this access can be implemented through the use of address arithmetic to combine a *base address* that specifies the beginning location of a vector in memory with an *index* that specifies the offset of a particular element of the vector.

Representing JavaScript data

We can use vectors to implement the basic pair structures required for a list-structured memory. Let us imagine that computer memory is divided into two vectors: `the_heads` and `the_tails`. We will represent list structure as follows: A pointer to a pair is an index into the two vectors. The head of the pair is the entry in `the_heads` with the designated index, and the tail of the pair is the entry in `the_tails` with the designated index. We also need a representation for objects other than pairs (such as numbers and strings) and a way to distinguish one kind of data from another. There are many methods of accomplishing this, but they all reduce to using *typed pointers*, that is, to extending the notion of 'pointer' to include information on data type.[6] The data type enables the system to distinguish a pointer to a pair (which consists of the 'pair' data type and an index into the memory vectors) from pointers to other kinds of data (which consist of some other data type and whatever is being used to represent data of that type). Two data objects are considered to be the same (`===`) if their pointers are identical.[7] Figure 5.14 illustrates the use of this method to represent the list `list(list(1, 2), 3, 4)`,

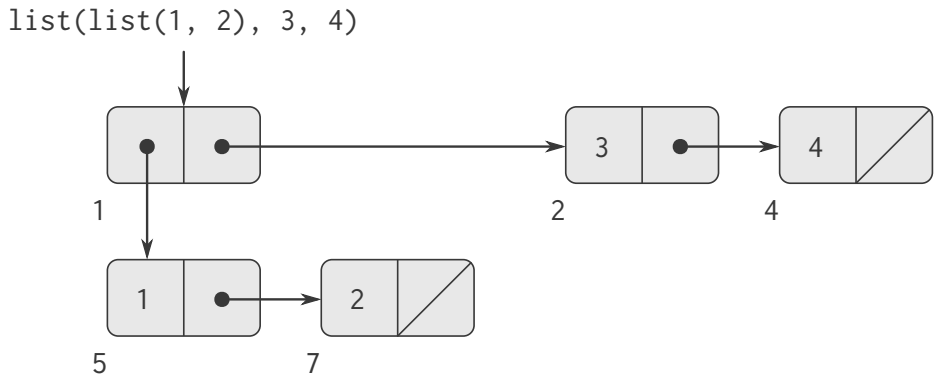
[4] We could represent memory as lists of items. However, the access time would then not be independent of the index, since accessing the *n*th element of a list requires $n - 1$ `tail` operations.

[5] For completeness, we should specify a `make_vector` operation that constructs vectors. However, in the present application we will use vectors only to model fixed divisions of the computer memory.

[6] This is precisely the same 'tagged data' idea we introduced in chapter 2 for dealing with generic operations. Here, however, the data types are included at the primitive machine level rather than constructed through the use of lists.

[7] Type information may be encoded in a variety of ways, depending on the details of the machine on which the JavaScript system is to be implemented. The execution efficiency of JavaScript programs will be strongly dependent on how cleverly this choice is made, but it is difficult to formulate general design rules for good choices. The most straightforward way to implement typed pointers is to allocate a fixed set of bits in each pointer to be a *type field* that encodes the data type. Important questions to be addressed in designing such a representation include the following: How many type bits are required? How large must the vector indices be? How efficiently can the primitive machine instructions be used to manipulate the type fields of pointers? Machines that include special hardware for the efficient handling of type fields are said to have *tagged architectures*.

whose box-and-pointer diagram is also shown. We use letter prefixes to denote the data-type information. Thus, a pointer to the pair with index 5 is denoted p5, the empty list is denoted by the pointer e0, and a pointer to the number 4 is denoted n4. In the box-and-pointer diagram, we have indicated at the lower left of each pair the vector index that specifies where the head and tail of the pair are stored. The blank locations in the_heads and the_tails may contain parts of other list structures (not of interest here).



Index	0	1	2	3	4	5	6	7	8	...
the_heads		p5	n3		n4	n1		n2		...
the_tails		p2	p4		e0	p7		e0		...

Figure 5.14: Box-and-pointer and memory-vector representations of the list list(list(1, 2), 3, 4).

A pointer to a number, such as n4, might consist of a type indicating numeric data together with the actual representation of the number 4.[8] To deal with numbers that are too large to be represented in the fixed amount of space allocated for a single pointer, we could use a distinct *bignum* data type, for which the pointer designates a list in which the parts of the number are stored.[9]

A string might be represented as a typed pointer that designates a sequence of the characters that form the string's printed representation. This sequence is constructed by the JavaScript

[8] This decision on the representation of numbers determines whether ===, which tests equality of pointers, can be used to test for equality of numbers. If the pointer contains the number itself, then equal numbers will have the same pointer. But if the pointer contains the index of a location where the number is stored, equal numbers will be guaranteed to have equal pointers only if we are careful never to store the same number in more than one location.

[9] This is just like writing a number as a sequence of digits, except that each 'digit' is a number between 0 and the largest number that can be stored in a single pointer.

reader when the character string is initially encountered in input. Since we want two instances of a string to be recognized as the 'same' string by === and we want === to be a simple test for equality of pointers, we must ensure that if the reader sees the same character string twice, it will use the same pointer (to the same sequence of characters) to represent both occurrences. To accomplish this, the reader maintains a table, traditionally called the *obarray*, of all the symbols it has ever encountered. When the reader encounters a character string and is about to construct a symbol, it checks the obarray to see if it has ever before seen the same character string. If it has not, it uses the characters to construct a new symbol (a typed pointer to a new character sequence) and enters this pointer in the obarray. If the reader has seen the string before, it returns the symbol pointer stored in the obarray. This process of replacing character strings by unique pointers is called *interning* symbols.

Implementing the primitive list operations

Given the above representation scheme, we can replace each 'primitive' list operation of a register machine with one or more primitive vector operations. We will use two registers, `the_heads` and `the_tails`, to identify the memory vectors, and will assume that `vector_ref` and `vector_set` are available as primitive operations. We also assume that numeric operations on pointers (such as incrementing a pointer, using a pair pointer to index a vector, or adding two numbers) use only the index portion of the typed pointer.

For example, we can make a register machine support the instructions

```
assign(reg,list(op("head"), reg))
assign(reg,list(op("tail"), reg))
```

if we implement these, respectively, as

```
assign(reg,list(op("vector_ref"), reg("the_heads"), reg))
assign(reg,list(op("vector_ref"), reg("the_tails"), reg))
```

The instructions

```
perform(list(op("set_head"), reg(), reg()))
perform(list(op("set_tail"), reg(), reg()))
```

are implemented as

```
perform(op("vector_set"), list(reg("the_heads"), reg(), reg()))
perform(op("vector_set"), list((reg("the_tails"), reg(), reg()))
```

The operation `pair` is performed by allocating an unused index and storing the arguments to `pair` in `the_heads` and `the_tails` at that indexed vector position. We presume that there is a special register, `free`, that always holds a pair pointer containing the next available index, and that we can increment the index part of that pointer to find the next free location.[10] For

[10]There are other ways of finding free storage. For example, we could link together all the unused pairs into a

example, the instruction

```
assign(, list(op("pair"), reg(), reg())
```

is implemented as the following sequence of vector operations:[11]

```
perform(op("vector_set"), list(reg("the_heads"), reg("free"), reg())),
perform(op("vector_set"), list(reg("the_tails"), reg("free"), reg())),
assign(, reg("free")),
assign("free", list(op("+"), reg("free"), cons(1)))
```

The === operation

```
list(op("==="), reg(), reg())
```

simply tests the equality of all fields in the registers, and predicates such as `is_pair`, `is_null`, `is_string`, and `is_number` need only check the type field.

Implementing stacks

Although our register machines use stacks, we need do nothing special here, since stacks can be modeled in terms of lists. The stack can be a list of the saved values, pointed to by a special register `the_stack`. Thus, `save(`*reg*`)` can be implemented as

```
assign("the_stack", list(op("pair"), reg(reg), reg("the-stack")))
```

Similarly, `restore(`*reg*`)` can be implemented as

```
assign(reg, list(op("head"), reg("the_stack")))
assign("the_stack", list(op("tail"), reg("the_stack")))
```

and `perform(op("initialize_stack"))` can be implemented as

```
assign("the_stack", cons(null))
```

These operations can be further expanded in terms of the vector operations given above. In conventional computer architectures, however, it is usually advantageous to allocate the stack as a separate vector. Then pushing and popping the stack can be accomplished by incrementing or decrementing an index into that vector.

Exercise 5.20

Draw the box-and-pointer representation and the memory-vector representation (as in Figure 5.14) of the list structure produced by

free list. Our free locations are consecutive (and hence can be accessed by incrementing a pointer) because we are using a compacting garbage collector, as we will see in section 5.3.2.

[11] This is essentially the implementation of `pair` in terms of `set_head` and `set_tail`, as described in section 3.3.1. The operation `get_new_pair` used in that implementation is realized here by the `free` pointer.

```
const x = pair(1, 2);
const y = list(x, x);
```

with the `free` pointer initially `p1`. What is the final value of `free`? What pointers represent the values of `x` and `y`?

Exercise 5.21

Implement register machines for the following functions. Assume that the list-structure memory operations are available as machine primitives.

a. Recursive `count_leaves`:

```
function count_leaves(tree) {
    return is_null(tree)
        ? 0
        : ! is_pair(tree)
          ? 1
          : count_leaves(head(tree)) +
            count_leaves(tail(tree));
}
```

b. Recursive `count_leaves` with explicit counter:

```
function count_leaves(tree) {
    function count_iter(tree, n) {
        return is_null(tree)
            ? n
            : ! is_pair(tree)
              ? n + 1
              : count_iter(tail(tree),
                           count_iter(head(tree), n)));
    }
    count_iter(tree, 0);
}
```

Exercise 5.22

Exercise 3.12 of section 3.3.1 presented an `append` function that appends two lists to form a new list and an `append_mutator` function that splices two lists together. Design a register machine to implement each of these functions. Assume that the list-structure memory operations are available as primitive operations.

5.3.2 Maintaining the Illusion of Infinite Memory

The representation method outlined in section 5.3.1 solves the problem of implementing list structure, provided that we have an infinite amount of memory. With a real computer we will eventually run out of free space in which to construct new pairs.[12] However, most of the pairs generated in a typical computation are used only to hold intermediate results. After these results are accessed, the pairs are no longer needed—they are *garbage*. For instance, the computation

```
accumulate((x, y) => x + y, 0, filter(is_odd, enumerate_interval(0, n)))
```

constructs two lists: the enumeration and the result of filtering the enumeration. When the accumulation is complete, these lists are no longer needed, and the allocated memory can be reclaimed. If we can arrange to collect all the garbage periodically, and if this turns out to recycle memory at about the same rate at which we construct new pairs, we will have preserved the illusion that there is an infinite amount of memory.

In order to recycle pairs, we must have a way to determine which allocated pairs are not needed (in the sense that their contents can no longer influence the future of the computation). The method we shall examine for accomplishing this is known as *garbage collection*. Garbage collection is based on the observation that, at any moment in a JavaScript interpretation, the only objects that can affect the future of the computation are those that can be reached by some succession of `head` and `tail` operations starting from the pointers that are currently in the machine registers.[13] Any memory cell that is not so accessible may be recycled.

There are many ways to perform garbage collection. The method we shall examine here is called *stop-and-copy*. The basic idea is to divide memory into two halves: 'working memory' and 'free memory.' When `pair` constructs pairs, it allocates these in working memory. When working memory is full, we perform garbage collection by locating all the useful pairs in working memory and copying these into consecutive locations in free memory. (The useful pairs are located by tracing all the `head` and `tail` pointers, starting with the machine registers.) Since we do not copy the garbage, there will presumably be additional free memory that we can use to allocate new pairs. In addition, nothing in the working memory is needed, since all the useful pairs in it have been copied. Thus, if we interchange the roles of working memory and free memory, we can continue processing; new pairs will be allocated in the new working

[12] This may not be true eventually, because memories may get large enough so that it would be impossible to run out of free memory in the lifetime of the computer. For example, there are about 3×10^{13}, microseconds in a year, so if we were to `pair` once per microsecond we would need about 10^{15} cells of memory to build a machine that could operate for 30 years without running out of memory. That much memory seems absurdly large by today's standards, but it is not physically impossible. On the other hand, processors are getting faster and a future computer may have large numbers of processors operating in parallel on a single memory, so it may be possible to use up memory much faster than we have postulated.

[13] We assume here that the stack is represented as a list as described in section 5.3.1, so that items on the stack are accessible via the pointer in the stack register.

memory (which was the old free memory). When this is full, we can copy the useful pairs into the new free memory (which was the old working memory).[14]

Implementation of a stop-and-copy garbage collector

We now use our register-machine language to describe the stop-and-copy algorithm in more detail. We will assume that there is a register called `root` that contains a pointer to a structure that eventually points at all accessible data. This can be arranged by storing the contents of all the machine registers in a pre-allocated list pointed at by `root` just before starting garbage collection.[15] We also assume that, in addition to the current working memory, there is free memory available into which we can copy the useful data. The current working memory consists of vectors whose base addresses are in registers called `the_heads` and `the_tails`, and the free memory is in registers called `new_heads` and `new_tails`.

Garbage collection is triggered when we exhaust the free cells in the current working memory, that is, when a `pair` operation attempts to increment the `free` pointer beyond the end of the memory vector. When the garbage-collection process is complete, the `root` pointer will point into the new memory, all objects accessible from the `root` will have been moved to the new memory, and the `free` pointer will indicate the next place in the new memory where a new pair can be allocated. In addition, the roles of working memory and new memory will have been interchanged—new pairs will be constructed in the new memory, beginning at the place indicated by `free`, and the (previous) working memory will be available as the new memory for the next garbage collection. Figure 5.15 shows the arrangement of memory just before and just after garbage collection.

[14] This idea was invented and first implemented by Minsky, as part of the implementation of Lisp for the PDP-1 at the MIT Research Laboratory of Electronics. It was further developed by Fenichel and Yochelson (1969) for use in the Lisp implementation for the Multics time-sharing system. Later, Baker (1978) developed a 'real-time' version of the method, which does not require the computation to stop during garbage collection. Baker's idea was extended by Hewitt, Lieberman, and Moon (see Lieberman and Hewitt 1983) to take advantage of the fact that some structure is more volatile and other structure is more permanent.

An alternative commonly used garbage-collection technique is the *mark-sweep* method. This consists of tracing all the structure accessible from the machine registers and marking each pair we reach. We then scan all of memory, and any location that is unmarked is 'swept up' as garbage and made available for reuse. A full discussion of the mark-sweep method can be found in Allen 1978.

The Minsky-Fenichel-Yochelson algorithm is the dominant algorithm in use for large-memory systems because it examines only the useful part of memory. This is in contrast to mark-sweep, in which the sweep phase must check all of memory. A second advantage of stop-and-copy is that it is a *compacting* garbage collector. That is, at the end of the garbage-collection phase the useful data will have been moved to consecutive memory locations, with all garbage pairs compressed out. This can be an extremely important performance consideration in machines with virtual memory, in which accesses to widely separated memory addresses may require extra paging operations.

[15] This list of registers does not include the registers used by the storage-allocation system—`root`, `the-heads`, `the-tails`, and the other registers that will be introduced in this section.

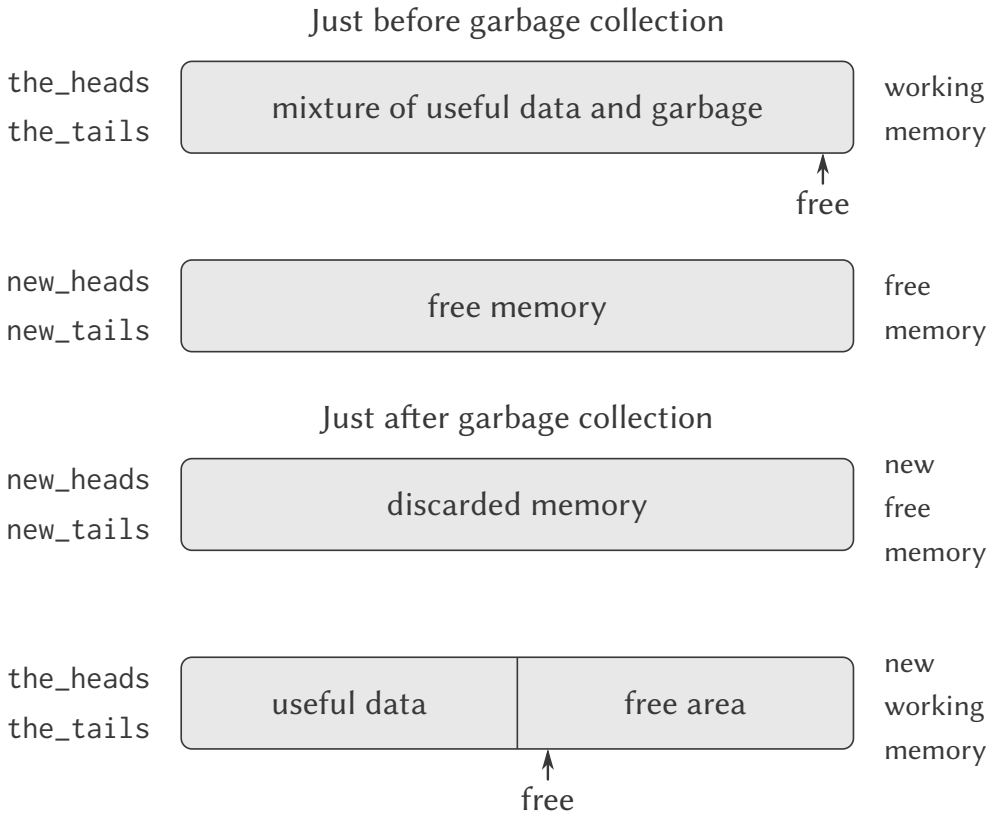


Figure 5.15: Reconfiguration of memory by the garbage-collection process.

The state of the garbage-collection process is controlled by maintaining two pointers: `free` and `scan`. These are initialized to point to the beginning of the new memory. The algorithm begins by relocating the pair pointed at by `root` to the beginning of the new memory. The pair is copied, the `root` pointer is adjusted to point to the new location, and the `free` pointer is incremented. In addition, the old location of the pair is marked to show that its contents have been moved. This marking is done as follows: In the `head` position, we place a special tag that signals that this is an already-moved object. (Such an object is traditionally called a *broken heart*.)[16] In the `tail` position we place a *forwarding address* that points at the location to which the object has been moved.

After relocating the root, the garbage collector enters its basic cycle. At each step in the algorithm, the `scan` pointer (initially pointing at the relocated root) points at a pair that has been moved to the new memory but whose `head` and `tail` pointers still refer to objects in the old memory. These objects are each relocated, and the `scan` pointer is incremented. To relocate an object (for example, the object indicated by the `head` pointer of the pair we are scanning) we check to see if the object has already been moved (as indicated by the presence of a broken-heart tag in the `head` position of the object). If the object has not already been moved, we copy it to the place indicated by `free`, update `free`, set up a broken heart at the

[16]The term *broken heart* was coined by David Cressey, who wrote a garbage collector for MDL, a dialect of JavaScript developed at MIT during the early 1970s.

object's old location, and update the pointer to the object (in this example, the head pointer of the pair we are scanning) to point to the new location. If the object has already been moved, its forwarding address (found in the tail position of the broken heart) is substituted for the pointer in the pair being scanned. Eventually, all accessible objects will have been moved and scanned, at which point the scan pointer will overtake the free pointer and the process will terminate.

We can specify the stop-and-copy algorithm as a sequence of instructions for a register machine. The basic step of relocating an object is accomplished by a subroutine called relocate_old_result_in_new. This subroutine gets its argument, a pointer to the object to be relocated, from a register named old. It relocates the designated object (incrementing free in the process), puts a pointer to the relocated object into a register called **new**, and returns by branching to the entry point stored in the register relocate-**continue**. To begin garbage collection, we invoke this subroutine to relocate the root pointer, after initializing free and scan. When the relocation of root has been accomplished, we install the new pointer as the new root and enter the main loop of the garbage collector.

```
"begin_garbage_collection",
    assign("free", constant(0)),
    assign("scan", constant(0)),
    assign("old", reg("root")),
    assign("relocate_continue", label("reassign_root")),
    go_to(label("relocate_old_result_in_new")),
"reassign_root",
    assign("root", reg("new")),
    go_to(label("gc_loop")),
```

In the main loop of the garbage collector we must determine whether there are any more objects to be scanned. We do this by testing whether the scan pointer is coincident with the free pointer. If the pointers are equal, then all accessible objects have been relocated, and we branch to gc-flip, which cleans things up so that we can continue the interrupted computation. If there are still pairs to be scanned, we call the relocate subroutine to relocate the head of the next pair (by placing the head pointer in old). The relocate_continue register is set up so that the subroutine will return to update the head pointer.

```
"gc_loop",
    test(list(op("==="), reg("scan"), reg("free"))),
    branch(label("gc_flip")),
    assign("old", list(op("vector_ref"), reg("new_heads"), reg("scan"))),
    assign("relocate_continue", label("update_head")),
    go_to(label("relocate_old_result_in_new")),
```

At update_head, we modify the head pointer of the pair being scanned, then proceed to relocate the tail of the pair. We return to update_tail when that relocation has been accomplished. After relocating and updating the tail, we are finished scanning that pair, so we continue

with the main loop.

```
"update_head",
     perform(list(op("vector_set"), reg("new_heads"), reg("scan"), reg("new"))),
     assign("old", list(op("vector_ref"), reg("new_tails"), reg("scan"))),
     assign("relocate_continue", label("update_tail")),
     go_to(label("relocate_old_result_in_new")),

"update_tail",
     perform(list(op("vector_set"), reg("new_tails"), reg("scan"), reg("new"))),
     assign("scan", list(op("+"), reg("scan"), cons(1))),
     go_to(label("gc_loop")),
```

The subroutine `relocate_old_result_in_new` relocates objects as follows: If the object to be relocated (pointed at by `old`) is not a pair, then we return the same pointer to the object unchanged (in **new**). (For example, we may be scanning a pair whose `head` is the number 4. If we represent the `head` by `n4`, as described in section 5.3.1, then we want the 'relocated' `head` pointer to still be `n4`.) Otherwise, we must perform the relocation. If the `head` position of the pair to be relocated contains a broken-heart tag, then the pair has in fact already been moved, so we retrieve the forwarding address (from the `tail` position of the broken heart) and return this in **new**. If the pointer in `old` points at a yet-unmoved pair, then we move the pair to the first free cell in new memory (pointed at by `free`) and set up the broken heart by storing a broken-heart tag and forwarding address at the old location. `Relocate_old_result_in_new` uses a register `oldhr` to hold the `head` or the `tail` of the object pointed at by `old`.[17]

```
"relocate_old_result_in_new",
    test(list(op("is_pointer_to_pair"), reg("old"))),
    branch(label("pair")),
    assign("new", reg("old")),
    go_to(reg("relocate_continue")),
"pair",
    assign("oldhr", list(op("vector_ref"), reg("the_heads"), reg("old"))),
    test(list(op("is_broken_heart"), reg("oldhr"))),
    branch(label("already_moved")),
    assign("new", reg("free")),       // new location for pair
    // Update "free" pointer.
    assign("free", list((op("+"), reg(free), cons(1))),
    // Copy the head and tail to new memory
    perform(list(op("vector_set"),
                 reg("new_heads"), reg("new"), reg("oldhr"))),
    assign("oldhr", list(op("vector_ref"), reg("the_tails"), reg("old"))),
    perform(list(op("vector_set"),
```

[17] The garbage collector uses the low-level predicate `is_pointer_to_pair` instead of the list-structure operation because in a real system there might be various things that are treated as pairs for garbage-collection purposes. For example, in a JavaScript system that conforms to the ECMAScript standard a function object may be implemented as a special kind of 'pair' that doesn't satisfy the `is_pair` predicate. For simulation purposes, `is_pointer_to_pair` can be implemented as `is_pair`.

```
                reg("new_tails"), reg("new"), reg("oldhr"))),
    // Construct the broken heart
    perform(list(op("vector_set"),
                reg("the_heads"), reg("old"), cons("broken_heart"))),
    perform(list(op("vector_set"),
                reg("the_tails"), reg("old"), reg("new"))),
    go_to(reg("relocate_continue")),
"already_moved",
    assign("new", list(op("vector_ref"), reg("the_tails"), reg("old"))),
    go_to(reg("relocate_continue")),
```

At the very end of the garbage collection process, we interchange the role of old and new memories by interchanging pointers: interchanging `the_heads` with `new_heads`, and `the_tails` with `new_tails`. We will then be ready to perform another garbage collection the next time memory runs out.

```
"gc_flip",
    assign("temp", reg("the_tails")),
    assign("the_tails", reg("new_tails")),
    assign("new_tails", reg("temp")),
    assign("temp", reg("the_heads")),
    assign("the_heads", reg("new_heads")),
    assign("new_heads", reg("temp")),
```

5.4 The Explicit-Control Evaluator

Note: this section is a work in progress!

In section 5.1 we saw how to transform simple Source programs into descriptions of register machines. We will now perform this transformation on a more complex program, the metacircular evaluator of sections 4.1.1–4.1.4, which shows how the behavior of a Source interpreter can be described in terms of the functions `eval` and `apply`. The *explicit-control evaluator* that we develop in this section shows how the underlying function-calling and argument-passing mechanisms used in the evaluation process can be described in terms of operations on registers and stacks. In addition, the explicit-control evaluator can serve as an implementation of a Source interpreter, written in a language that is very similar to the native machine language of conventional computers. The evaluator can be executed by the register-machine simulator of section 5.2. Alternatively, it can be used as a starting point for building a machine-language implementation of a Source evaluator, or even a special-purpose machine for evaluating Source expressions. Figure 5.16 shows such a hardware implementation: a silicon chip that acts as an evaluator for Scheme, the language for which this book was originally written.. The chip designers started with the data-path and controller specifications for a register machine similar

to the evaluator described in this section and used design automation programs to construct the integrated-circuit layout.[18]

Registers and operations

In designing the explicit-control evaluator, we must specify the operations to be used in our register machine. We described the metacircular evaluator in terms of abstract syntax, using functions such as `quoted?` and `make-procedure`. In implementing the register machine, we could expand these functions into sequences of elementary list-structure memory operations, and implement these operations on our register machine. However, this would make our evaluator very long, obscuring the basic structure with details. To clarify the presentation, we will include as primitive operations of the register machine the syntax functions given in section 4.1.2 and the functions for representing environments and other run-time data given in sections 4.1.3 and 4.1.4. In order to completely specify an evaluator that could be programmed in a low-level machine language or implemented in hardware, we would replace these operations by more elementary operations, using the list-structure implementation we described in section 5.3.

[18] See Batali et al. 1982 for more information on the chip and the method by which it was designed.

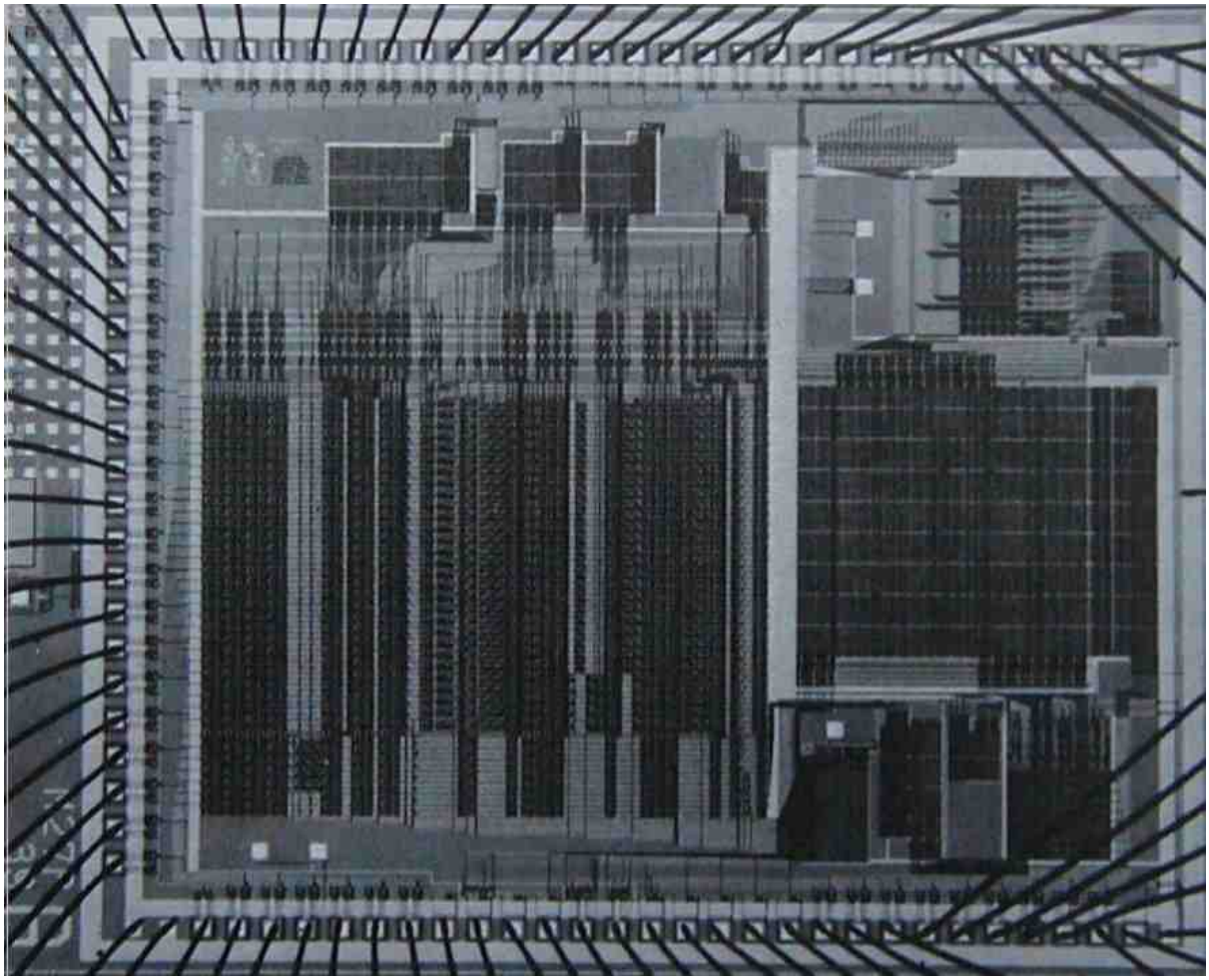


Figure 5.16: A silicon-chip implementation of an evaluator for Scheme.

Our Source evaluator register machine includes a stack and seven registers: `exp`, `env`, `val`, **`continue`**, `proc`, `argl`, and `unev`. `Exp` is used to hold the expression to be evaluated, and `env` contains the environment in which the evaluation is to be performed. At the end of an evaluation, `val` contains the value obtained by evaluating the expression in the designated environment. The **`continue`** register is used to implement recursion, as explained in section 5.1.4. (The evaluator needs to call itself recursively, since evaluating an expression requires evaluating its subexpressions.) The registers `proc`, `argl`, and `unev` are used in evaluating combinations.

We will not provide a data-path diagram to show how the registers and operations of the evaluator are connected, nor will we give the complete list of machine operations. These are implicit in the evaluator's controller, which will be presented in detail.

5.4.1 The Core of the Explicit-Control Evaluator

The central element in the evaluator is the sequence of instructions beginning at `eval_dispatch`. This corresponds to the `evaluate` function of the metacircular evaluator described in section 4.1.1. When the controller starts at `eval_dispatch`, it evaluates the expression specified by `exp` in the environment specified by `env`. When evaluation is complete, the controller will go to the entry point stored in **`continue`**, and the `val` register will hold the value of the expression. As with the metacircular `eval`, the structure of `eval_dispatch` is a case analysis on the syntactic type of the expression to be evaluated.[19]

```
"eval_dispatch",
          test(op("is_self_evaluating"), reg("exp")),
          branch(label("ev_self_eval")),
          test(op("is_variable"), reg("exp")),
          branch(label("ev_variable")),
          test(op("is_quoted"), reg("exp")),
          branch(label("ev_quoted")), /// FIXME
          test(op("is_assignment"), reg("exp")),
          branch(label("ev_assignment")),
          test(op("is_definition"), reg("exp")),
          branch(label("ev_definition")),
          test(op("is_conditional_statement"), reg("exp")),
          branch(label("ev_if")),
          test(op("is_function_expression"), reg("exp")),
          branch(label("ev_lambda")),
          test(op("is_block"), reg("exp")),
          branch(label("ev_begin")),
          test(op("is_application"), reg("exp")),
          branch(label("ev_application")),
          go_to(label("unknown_expression_type")),
```

Evaluating simple expressions

Numbers and strings (which are self-evaluating), variables, quotations, and **`function`** `definition` expressions have no subexpressions to be evaluated. For these, the evaluator simply places the correct value in the `val` register and continues execution at the entry point specified by **`continue`**. Evaluation of simple expressions is performed by the following controller code:

```
"ev_self_eval",
    assign("val", reg("exp")),
    go_to(reg("continue")),
```

[19] In our controller, the dispatch is written as a sequence of `test` and `branch` instructions. Alternatively, it could have been written in a data-directed style (and in a real system it probably would have been) to avoid the need to perform sequential tests and to facilitate the definition of new expression types. A machine designed to run JavaScript would probably include a `dispatch-on-type` instruction that would efficiently execute such data-directed dispatches.

```
"ev_variable",
    assign("val", op("lookup_variable_value"), reg("exp"), reg("env")),
    go_to(reg("continue")),

"ev_quoted",
    assign("val", op("text_of_quotation"), reg("exp")),
    go_to(reg("continue")),

"ev_lambda",
    assign("unev", op("lambda_parameters"), reg("exp")),
    assign("exp", op("lambda_body"), reg("exp")),
    assign("val", op("make_procedure"), reg("unev"), reg("exp"), reg("env")),
    go_to(reg("continue")),
```

Observe how `ev_lambda` uses the `unev` and `exp` registers to hold the parameters and body of the lambda expression so that they can be passed to the `make_procedure` operation, along with the environment in `env`.

Evaluating function applications

A function application is specified by a combination containing an operator and operands. The operator is a subexpression whose value is a function, and the operands are subexpressions whose values are the arguments to which the function should be applied. The metacircular `eval` handles applications by calling itself recursively to evaluate each element of the combination, and then passing the results to `apply`, which performs the actual function application. The explicit-control evaluator does the same thing; these recursive calls are implemented by `goto` instructions, together with use of the stack to save registers that will be restored after the recursive call returns. Before each call we will be careful to identify which registers must be saved (because their values will be needed later).[20]

We begin the evaluation of an application by evaluating the operator to produce a function, which will later be applied to the evaluated operands. To evaluate the operator, we move it to the `exp` register and go to `eval-dispatch`. The environment in the `env` register is already the correct one in which to evaluate the operator. However, we save `env` because we will need it later to evaluate the operands. We also extract the operands into `unev` and save this on the stack. We set up **`continue`** so that `eval-dispatch` will resume at `ev-appl-did-operator` after the operator has been evaluated. First, however, we save the old value of **`continue`**, which tells

[20] This is an important but subtle point in translating algorithms from a procedural language, such as JavaScript, to a register-machine language. As an alternative to saving only what is needed, we could save all the registers (except `val`) before each recursive call. This is called a *framed-stack* discipline. This would work but might save more registers than necessary; this could be an important consideration in a system where stack operations are expensive. Saving registers whose contents will not be needed later may also hold onto useless data that could otherwise be garbage-collected, freeing space to be reused.

the controller where to continue after the application.

```
"ev_application",
          save("continue"),
          save("env"),
          assign("unev", op("operands"), reg("exp")),
          save("unev"),
          assign("exp", op("operator"), reg("exp")),
          assign("continue", label("ev_appl_did_operator")),
          go_to(label("eval_dispatch")),
```

Upon returning from evaluating the operator subexpression, we proceed to evaluate the operands of the combination and to accumulate the resulting arguments in a list, held in `argl`. First we restore the unevaluated operands and the environment. We initialize `argl` to an empty list. Then we assign to the `proc` register the function that was produced by evaluating the operator. If there are no operands, we go directly to `apply-dispatch`. Otherwise we save `proc` on the stack and start the argument-evaluation loop:[21]

```
"ev_appl_did_operator",
           restore("unev"),                  // the operands
           restore("env"),
           assign("argl", op("empty_arglist")),
           assign("fun", reg("val")),        // the operator
           test(op("has_no_operands"), reg("unev")),
           branch(label("apply_dispatch")),
           save("fun"),
```

Each cycle of the argument-evaluation loop evaluates an operand from the list in `unev` and accumulates the result into `argl`. To evaluate an operand, we place it in the `exp` register and go to `eval_dispatch`, after setting **`continue`** so that execution will resume with the argument-accumulation phase. But first we save the arguments accumulated so far (held in `argl`), the environment (held in `env`), and the remaining operands to be evaluated (held in `unev`). A special case is made for the evaluation of the last operand, which is handled at `ev_appl_last_arg`.

```
"ev_appl_operand_loop",
```

[21] We add to the evaluator data-structure functions in section 4.1.3 the following two functions for manipulating argument lists:

```
const empty_arglist = list();


function adjoin_arg(arg, arglist) {
    return append(arglist, list(arg));
}
```

We also use an additional syntax function to test for the last operand in a combination:

```
function is_last_operand(ops) {
    return is_null(tail(ops));
}
```

```
            save("argl"),
            assign("exp", (op("first_operand"), reg("unev")),
            test(op("is_last_operand"), reg("unev")),
            branch(label("ev_appl_last_arg")),
            save("env"),
            save("unev"),
            assign("continue", (label("ev_appl_accumulate_arg"))),
            go_to(label("eval_dispatch")),
```

When an operand has been evaluated, the value is accumulated into the list held in `argl`. The operand is then removed from the list of unevaluated operands in `unev`, and the argument-evaluation continues.

```
"ev_appl_accumulate_arg",
          restore("unev"),
          restore("env"),
          restore("argl"),
          assign("argl", op("adjoin_arg"), reg("val"), reg("argl")),
          assign("unev", op("rest_operands"), reg("unev")),
          go_to(label("ev_appl_operand_loop")),
```

Evaluation of the last argument is handled differently. There is no need to save the environment or the list of unevaluated operands before going to `eval-dispatch`, since they will not be required after the last operand is evaluated. Thus, we return from the evaluation to a special entry point `ev-appl-accum-last-arg`, which restores the argument list, accumulates the new argument, restores the saved function, and goes off to perform the application.[22]

```
"ev_appl_last_arg",
      assign("continue", label("ev_appl_accum_last_arg")),
      go_to(label("eval_dispatch")),

"ev_appl_accum_last_arg",
      restore(argl),
      assign("argl", op("adjoin_arg"), reg("val"), reg("argl")),
      restore("fun"),
      go_to(label("apply_dispatch")),
```

The details of the argument-evaluation loop determine the order in which the interpreter evaluates the operands of a combination (e.g., left to right or right to left—see exercise 3.8). This order is not determined by the metacircular evaluator, which inherits its control structure from the underlying Scheme in which it is implemented.[23] Because the `first\?operand` selector

[22] The optimization of treating the last operand specially is known as *evlis tail recursion* (see Wand 1980). We could be somewhat more efficient in the argument evaluation loop if we made evaluation of the first operand a special case too. This would permit us to postpone initializing `argl` until after evaluating the first operand, so as to avoid saving `argl` in this case. The compiler in section 5.5 performs this optimization. (Compare the `construct-arglist` function of section 5.5.3.)

[23] The order of operand evaluation in the metacircular evaluator is determined by the order of evaluation of the arguments to `pair` in the function `list-of-values` of section 4.1.1 (see exercise 4.1).

(used in ev-appl-operand-loop to extract successive operands from unev) is implemented as head and the rest-operands selector is implemented as tail, the explicit-control evaluator will evaluate the operands of a combination in left-to-right order.

Function application

The entry point apply-dispatch corresponds to the apply function of the metacircular evaluator. By the time we get to apply-dispatch, the proc register contains the function to apply and argl contains the list of evaluated arguments to which it must be applied. The saved value of **continue** (originally passed to eval-dispatch and saved at ev-application), which tells where to return with the result of the function application, is on the stack. When the application is complete, the controller transfers to the entry point specified by the saved **continue**, with the result of the application in val. As with the metacircular apply, there are two cases to consider. Either the function to be applied is a primitive or it is a compound function.

```
"apply_dispatch",
          test(op("is_primitive_procedure"), reg("fun")),
          branch(label("primitive_apply")),
          test(op("is_compound_procedure"), reg("fun")),
          branch(label("compound_apply")),
          go_to(label("unknown_procedure_type")),
```

We assume that each primitive is implemented so as to obtain its arguments from argl and place its result in val. To specify how the machine handles primitives, we would have to provide a sequence of controller instructions to implement each primitive and arrange for primitive_apply to dispatch to the instructions for the primitive identified by the contents of fun. Since we are interested in the structure of the evaluation process rather than the details of the primitives, we will instead just use an apply-primitive-procedure operation that applies the function in proc to the arguments in argl. For the purpose of simulating the evaluator with the simulator of section 5.2 we use the function apply-primitive-procedure, which calls on the underlying Scheme system to perform the application, just as we did for the metacircular evaluator in section 4.1.4. After computing the value of the primitive application, we restore **continue** and go to the designated entry point.

```
"primitive_apply",
          assign("val", op("apply_primitive_procedure"), reg("fun"), reg("argl")),
          restore("continue"),
          go_to(reg("continue")),
```

To apply a compound function, we proceed just as with the metacircular evaluator. We construct a frame that binds the function's parameters to the arguments, use this frame to extend the environment carried by the function, and evaluate in this extended environment the sequence of expressions that forms the body of the function. Ev-sequence, described below in

section 5.4.2, handles the evaluation of the sequence.

```
"compound_apply",
          assign("unev", op("procedure_parameters"), reg("fun")),
          assign("env", op("procedure_environment") reg("fun")),
          assign("env", op("extend_environment"), reg("unev"), reg("argl"), reg("env
          assign("unev", op("procedure_body"), reg("fun")),
          go_to(label("ev_sequence")),
```

`Compound-apply` is the only place in the interpreter where the `env` register is ever assigned a new value. Just as in the metacircular evaluator, the new environment is constructed from the environment carried by the function, together with the argument list and the corresponding list of variables to be bound.

5.4.2 Sequence Evaluation and Tail Recursion

The portion of the explicit-control evaluator at `ev-sequence` is analogous to the metacircular evaluator's `eval_sequence` function. It handles sequences of expressions in function bodies or in explicit `begin` expressions.

Explicit `begin` expressions are evaluated by placing the sequence of expressions to be evaluated in `unev`, saving **`continue`** on the stack, and jumping to `ev_sequence`.

The implicit sequences in function bodies are handled by jumping to `ev_sequence` from `compound_apply`, at which point **`continue`** is already on the stack, having been saved at `ev_application`.

The entries at `ev_sequence` and `ev_sequence_continue` form a loop that successively evaluates each expression in a sequence. The list of unevaluated expressions is kept in `unev`. Before evaluating each expression, we check to see if there are additional expressions to be evaluated in the sequence. If so, we save the rest of the unevaluated expressions (held in `unev`) and the environment in which these must be evaluated (held in `env`) and call `eval_dispatch` to evaluate the expression. The two saved registers are restored upon the return from this evaluation, at `ev_sequence_continue`.

The final expression in the sequence is handled differently, at the entry point `ev_sequence_last_exp`. Since there are no more expressions to be evaluated after this one, we need not save `unev` or `env` before going to `eval_dispatch`. The value of the whole sequence is the value of the last expression, so after the evaluation of the last expression there is nothing left to do except continue at the entry point currently held on the stack (which was saved by `ev_application` or `ev_begin`.) Rather than setting up **`continue`** to arrange for `eval_dispatch` to return here and then restoring **`continue`** from the stack and continuing at that entry point, we restore **`continue`** from the stack before going to `eval_dispatch`, so that `eval_dispatch` will continue at that entry point after evaluating the expression.

```
"ev_sequence",
```

```
        assign(exp(op("first_exp"), reg("unev"))),
        test(op("is_last_exp"), reg("unev")),
        branch(label("ev_sequence_last_exp")),
        save("unev"),
        save("env"),
        assign("continue", label("ev_sequence_continue")),
        go_to(label("eval_dispatch")),
    "ev_sequence_continue",
        restore("env"),
        restore("unev"),
        assign("unev", op("rest_exps"), reg("unev")),
        go_to(label("ev_sequence")),
    "ev_sequence_last_exp",
        restore("continue"),
        go_to(label("eval_dispatch")),
```

Tail recursion

In chapter 1 we said that the process described by a function such as

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

is an iterative process. Even though the function is syntactically recursive (defined in terms of itself), it is not logically necessary for an evaluator to save information in passing from one call to `sqrt_iter` to the next.[24] An evaluator that can execute a function such as `sqrt_iter` without requiring increasing storage as the function continues to call itself is called a *tail-recursive* evaluator. The metacircular implementation of the evaluator in chapter 4 does not specify whether the evaluator is tail-recursive, because that evaluator inherits its mechanism for saving state from the underlying Scheme. With the explicit-control evaluator, however, we can trace through the evaluation process to see when function calls cause a net accumulation of information on the stack.

Our evaluator is tail-recursive, because in order to evaluate the final expression of a sequence we transfer directly to `eval_dispatch` without saving any information on the stack. Hence, evaluating the final expression in a sequence—even if it is a function call (as in `sqrt_iter`, where the conditional expression, which is the last expression in the function body, reduces to a call to `sqrt_iter`)—will not cause any information to be accumulated on the stack.[25]

[24] We saw in section 5.1 how to implement such a process with a register machine that had no stack; the state of the process was stored in a fixed set of registers.

[25] This implementation of tail recursion in `ev_sequence` is one variety of a well-known optimization technique used by many compilers. In compiling a function that ends with a function call, one can replace the call by a jump

If we did not think to take advantage of the fact that it was unnecessary to save information in this case, we might have implemented `eval_sequence` by treating all the expressions in a sequence in the same way—saving the registers, evaluating the expression, returning to restore the registers, and repeating this until all the expressions have been evaluated:[26]

```
"ev_sequence",
          test(op("has_no_more_exps"), reg("unev")),
          branch(label("ev_sequence_end")),
          assign(exp(op("first_exp"), reg("unev")),
          save("unev"),
          save("env"),
          assign(continue(label("ev_sequence_continue"))),
          go_to(label("eval_dispatch")),
      "ev_sequence_continue",
          restore("env"),
          restore("unev"),
          assign("unev", op("rest_exps"), reg("unev")),
          go_to(label("ev_sequence")),
      "ev_sequence_end",
          restore("continue"),
          go_to(reg("continue")),
```

This may seem like a minor change to our previous code for evaluation of a sequence: The only difference is that we go through the save-restore cycle for the last expression in a sequence as well as for the others. The interpreter will still give the same value for any expression. But this change is fatal to the tail-recursive implementation, because we must now return after evaluating the final expression in a sequence in order to undo the (useless) register saves. These extra saves will accumulate during a nest of function calls. Consequently, processes such as `sqrt_iter` will require space proportional to the number of iterations rather than requiring constant space. This difference can be significant. For example, with tail recursion, an infinite loop can be expressed using only the function-call mechanism:

```
function count(n) {
          display(n, "\n");
          count(n + 1);
      }
```

Without tail recursion, such a function would eventually run out of stack space, and expressing a true iteration would require some control mechanism other than function call.

---

to the called function's entry point. Building this strategy into the interpreter, as we have done in this section, provides the optimization uniformly throughout the language.

[26] We can define `has_no_more_exps` as follows:

```
function has_no_more_exps(seq) {
          return is_null(seq);
       }
```

5.4.3 Conditionals, Assignments, and Definitions

As with the metacircular evaluator, special forms are handled by selectively evaluating fragments of the expression. For an **`if`** expression, we must evaluate the predicate and decide, based on the value of predicate, whether to evaluate the consequent or the alternative.

Before evaluating the predicate, we save the **`if`** expression itself so that we can later extract the consequent or alternative. We also save the environment, which we will need later in order to evaluate the consequent or the alternative, and we save **`continue`**, which we will need later in order to return to the evaluation of the expression that is waiting for the value of the **`if`**.

```
"ev_if",
          save("exp"),                    // save expression for later
          save("env"),
          save("continue"),
          assign("continue", label("ev_if_decide")),
          assign("exp", op("if_predicate"), reg("exp")),
          go_to(label("eval_dispatch")),  // evaluate the predicate
```

When we return from evaluating the predicate, we test whether it was true or false and, depending on the result, place either the consequent or the alternative in `exp` before going to `eval_dispatch`. Notice that restoring `env` and **`continue`** here sets up `eval_dispatch` to have the correct environment and to continue at the right place to receive the value of the **`if`** expression.

```
"ev_if_decide",
          restore("continue"),
          restore("env"),
          restore("exp"),
          test(op("is_true"), reg("val")),
          branch(label("ev_if_consequent")),

      "ev_if_alternative",
          assign("exp", op("if_alternative"), reg("exp")),
          go_to(label("eval_dispatch")),

      "ev_if_consequent",
          assign("exp", op("if_consequent"), reg("exp")),
          go_to(label("eval_dispatch")),
```

Assignments and definitions

Assignments are handled by `ev_assignment`, which is reached from `eval_dispatch` with the assignment expression in `exp`. The code at `ev_assignment` first evaluates the value part of the expression and then installs the new value in the environment. `Set_variable_value` is assumed to be available as a machine operation.

```
"ev_assignment",
          assign("unev", op("assignment_variable"), reg("exp")),
          save("unev"),                   // save variable for later
          assign("exp", op("assignment_value"), reg("exp")),
          save("env"),
          save("continue"),
          assign("continue", label("ev_assignment_1")),
          go_to(label("eval_dispatch")),  // evaluate the assignment value

      "ev_assignment_1",
          restore("continue"),
          restore("env"),
          restore("unev"),
          perform(op("set_variable_value"), reg("unev"), reg("val"), reg("env")),
          assign("val", const("ok")),
          go_to(reg("continue")),
```

Definitions are handled in a similar way:

```
"ev-definition",
          assign("unev", op("definition-variable"), reg("exp")),
          save("unev"),                   // save variable for later
          assign("exp", op("definition-value"), reg("exp")),
          save("env"),
          save("continue"),
          assign("continue", label("ev-definition-1")),
          go_to(label("eval-dispatch")),  // evaluate the definition value

      "ev-definition-1",
          restore("continue"),
          restore("env"),
          restore("unev"),
          perform(op("define-variable"), reg("unev"), reg("val"), reg("env")),
          assign("val", const("ok")),
          go_to(reg("continue")),
```

Exercise 5.23

Extend the evaluator to handle derived expressions such as `cond`, **`let`**, and so on (section 4.1.2). You may 'cheat' and assume that the syntax transformers such as `cond->`**`if`** are available as

machine operations.[27]

Exercise 5.24

Implement cond as a new basic special form without reducing it to **`if`**. You will have to construct a loop that tests the predicates of successive cond clauses until you find one that is true, and then use `ev-sequence` to evaluate the actions of the clause.

Exercise 5.25

Modify the evaluator so that it uses normal-order evaluation, based on the lazy evaluator of section 4.2.

5.4.4 Running the Evaluator

With the implementation of the explicit-control evaluator we come to the end of a development, begun in chapter 1, in which we have explored successively more precise models of the evaluation process. We started with the relatively informal substitution model, then extended this in chapter 3 to the environment model, which enabled us to deal with state and change. In the metacircular evaluator of chapter 4, we used Scheme itself as a language for making more explicit the environment structure constructed during evaluation of an expression. Now, with register machines, we have taken a close look at the evaluator's mechanisms for storage management, argument passing, and control. At each new level of description, we have had to raise issues and resolve ambiguities that were not apparent at the previous, less precise treatment of evaluation. To understand the behavior of the explicit-control evaluator, we can simulate it and monitor its performance.

We will install a driver loop in our evaluator machine. This plays the role of the `driver_loop` function of section 4.1.4. The evaluator will repeatedly print a prompt, read an expression, evaluate the expression by going to `eval_dispatch`, and print the result. The following instructions form the beginning of the explicit-control evaluator's controller sequence:[28]

[27] This isn't really cheating. In an actual implementation built from scratch, we would use our explicit-control evaluator to interpret a JavaScript program that performs source-level transformations in a syntax phase that runs before execution.

[28] We assume here that `read` and the various printing operations are available as primitive machine operations, which is useful for our simulation, but completely unrealistic in practice. These are actually extremely complex operations. In practice, they would be implemented using low-level input-output operations such as transferring single characters to and from a device.

To support the `get_global_environment` operation we define

```
const the_global_environment = setup_environment();
```

```
"read_eval_print_loop",
          perform(op("initialize_stack")),
          perform(op("prompt_for_input"), const("/// EC_Eval input:")),
          assign("exp", op("read")),
          assign("env", op("get_global_environment")),
          assign("continue", label("print_result")),
          go_to(label("eval_dispatch")),
          <!__  \indcode*{print_result} __>
      "print_result",
          perform(op("announce_output"), const(";;; EC_Eval value:")),
          perform(op("user_print"), reg("val")),
          go_to(label("read_eval_print_loop")),
```

When we encounter an error in a function (such as the 'unknown function type error' indicated at `apply_dispatch`), we print an error message and return to the driver loop.[29]

```
"unknown_expression_type",
          assign("val", const("unknown_expression_type_error")),
          go_to(label("signal_error")),


      "unknown_procedure_type",
          restore("continue"),    /// clean up stack (from apply_dispatch)
          assign(val(const("unknown_procedure_type_error"))),
          go_to(label("signal_error")),


      "signal_error",
          perform(op("user_print"), reg("val")),
          go_to(label("read_eval_print_loop")),
```

For the purposes of the simulation, we initialize the stack each time through the driver loop, since it might not be empty after an error (such as an undefined variable) interrupts an evaluation.[30]

If we combine all the code fragments presented in sections 5.4.1–5.4.4, we can create an evaluator machine model that we can run using the register-machine simulator of section 5.2.

```
function eceval() {
    return make_machine(list("exp", "env", "val", "proc", "argl", "continue", "unev"
                        eceval_operations,
```

```
function get_global_environment() {
    return the_global_environment;
}
```

[29] There are other errors that we would like the interpreter to handle, but these are not so simple. See exercise 5.30.

[30] We could perform the stack initialization only after errors, but doing it in the driver loop will be convenient for monitoring the evaluator's performance, as described below.

```
                        list(read_eval_print_loop,
                             ... /* entire machine controller as given above */
                             ));
}
```

We must define Scheme functions to simulate the operations used as primitives by the evaluator. These are the same functions we used for the metacircular evaluator in section 4.1, together with the few additional ones defined in footnotes throughout section 5.4.

```
const eceval_operations =
    list(list("is_self_evaluating", is_self_evaluating),
         ... /* complete list of operations for eceval machine */
         );
```

Finally, we can initialize the global environment and run the evaluator:

```
const the_global_environment = setup_environment();

start(eceval);
```

Of course, evaluating expressions in this way will take much longer than if we had directly typed them into Scheme, because of the multiple levels of simulation involved. Our expressions are evaluated by the explicit-control-evaluator machine, which is being simulated by a Scheme program, which is itself being evaluated by the Scheme interpreter.

Monitoring the performance of the evaluator

Simulation can be a powerful tool to guide the implementation of evaluators. Simulations make it easy not only to explore variations of the register-machine design but also to monitor the performance of the simulated evaluator. For example, one important factor in performance is how efficiently the evaluator uses the stack. We can observe the number of stack operations required to evaluate various expressions by defining the evaluator register machine with the version of the simulator that collects statistics on stack use (section 5.2.4), and adding an instruction at the evaluator's `print_result` entry point to print the statistics:

```
"print_result",
          perform(op("print_stack_statistics")), // added instruction
          perform(op("announce_output"), const("/// EC-Eval value:")),
          /* ... same as before ... */
```

Interactions with the evaluator now look like this:

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

```
(total-pushes = 3 maximum-depth = 3)
/// EC-Eval value:
ok

/// EC-Eval input:
factorial(5);
(total-pushes = 144 maximum-depth = 28)
/// EC-Eval value:
120
```

Note that the driver loop of the evaluator reinitializes the stack at the start of each interaction, so that the statistics printed will refer only to stack operations used to evaluate the previous expression.

Exercise 5.26

Use the monitored stack to explore the tail-recursive property of the evaluator (section 5.4.2). Start the evaluator and define the iterative `factorial` function from section **??**:

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

    return iter(1, 1, n);
}
```

Run the function with some small values of n. Record the maximum stack depth and the number of pushes required to compute $n!$ for each of these values.

a. You will find that the maximum depth required to evaluate $n!$ is independent of n. What is that depth?

b. Determine from your data a formula in terms of n for the total number of push operations used in evaluating $n!$ for any $n \geq 1$. Note that the number of operations used is a linear function of n and is thus determined by two constants.

Exercise 5.27

For comparison with exercise 5.26, explore the behavior of the following function for computing factorials recursively:

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

By running this function with the monitored stack, determine, as a function of n, the maximum depth of the stack and the total number of pushes used in evaluating $n!$ for $n \geq 1$. (Again, these functions will be linear.) Summarize your experiments by filling in the following table with the appropriate expressions in terms of n:

	Maximum depth	Number of pushes
Recursive factorial		
Iterative factorial		

The maximum depth is a measure of the amount of space used by the evaluator in carrying out the computation, and the number of pushes correlates well with the time required.

Exercise 5.28

Modify the definition of the evaluator by changing `eval_sequence` as described in section 5.4.2 so that the evaluator is no longer tail-recursive. Rerun your experiments from exercises 5.26 and 5.27 to demonstrate that both versions of the `factorial` function now require space that grows linearly with their input.

Exercise 5.29

Monitor the stack operations in the tree-recursive Fibonacci computation:

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

a. Give a formula in terms of n for the maximum depth of the stack required to compute Fib(n) for $n \geq 2$. Hint: In section **??** we argued that the space used by this process grows linearly with n.

b. Give a formula for the total number of pushes used to compute Fib(n) for $n \geq 2$. You should find that the number of pushes (which correlates well with the time used) grows exponentially with n. Hint: Let $S(n)$ be the number of pushes used in computing Fib(n). You should be able to argue that there is a formula that expresses $S(n)$ in terms of $S(n-1)$,

$S(n-2)$, and some fixed 'overhead' constant k that is independent of n. Give the formula, and say what k is. Then show that $S(n)$ can be expressed as $a\text{Fib}(n+1)+b$ and give the values of a and b.

Exercise 5.30

Our evaluator currently catches and signals only two kinds of errors—unknown expression types and unknown function types. Other errors will take us out of the evaluator read-eval-print loop. When we run the evaluator using the register-machine simulator, these errors are caught by the underlying Scheme system. This is analogous to the computer crashing when a user program makes an error.[31] It is a large project to make a real error system work, but it is well worth the effort to understand what is involved here.

a. Errors that occur in the evaluation process, such as an attempt to access an unbound variable, could be caught by changing the lookup operation to make it return a distinguished condition code, which cannot be a possible value of any user variable. The evaluator can test for this condition code and then do what is necessary to go to `signal-error`. Find all of the places in the evaluator where such a change is necessary and fix them. This is lots of work.

b. Much worse is the problem of handling errors that are signaled by applying primitive functions, such as an attempt to divide by zero or an attempt to extract the `head` of a symbol. In a professionally written high-quality system, each primitive application is checked for safety as part of the primitive. For example, every call to `head` could first check that the argument is a pair. If the argument is not a pair, the application would return a distinguished condition code to the evaluator, which would then report the failure. We could arrange for this in our register-machine simulator by making each primitive function check for applicability and returning an appropriate distinguished condition code on failure. Then the `primitive-apply` code in the evaluator can check for the condition code and go to `signal-error` if necessary. Build this structure and make it work. This is a major project.

[31] Regrettably, this is the normal state of affairs in conventional compiler-based language systems such as C. In UNIX™ the system 'dumps core,' and in DOS/Windows™ it becomes catatonic. The Macintosh™ displays a picture of an exploding bomb and offers you the opportunity to reboot the computer—if you're lucky.

5.5 Compilation

Note: this section is a work in progress!

The explicit-control evaluator of section 5.4 is a register machine whose controller interprets JavaScript programs. In this section we will see how to run JavaScript programs on a register machine whose controller is not a JavaScript interpreter.

The explicit-control evaluator machine is universal—it can carry out any computational process that can be described in JavaScript. The evaluator's controller orchestrates the use of its data paths to perform the desired computation. Thus, the evaluator's data paths are universal: They are sufficient to perform any computation we desire, given an appropriate controller.[32]

Commercial general-purpose computers are register machines organized around a collection of registers and operations that constitute an efficient and convenient universal set of data paths. The controller for a general-purpose machine is an interpreter for a register-machine language like the one we have been using. This language is called the *native language* of the machine, or simply *machine language*. Programs written in machine language are sequences of instructions that use the machine's data paths. For example, the explicit-control evaluator's instruction sequence can be thought of as a machine-language program for a general-purpose computer rather than as the controller for a specialized interpreter machine.

There are two common strategies for bridging the gap between higher-level languages and register-machine languages. The explicit-control evaluator illustrates the strategy of interpretation. An interpreter written in the native language of a machine configures the machine to execute programs written in a language (called the *source language*) that may differ from the native language of the machine performing the evaluation. The primitive functions of the source language are implemented as a library of subroutines written in the native language of the given machine. A program to be interpreted (called the *source program*) is represented as a data structure. The interpreter traverses this data structure, analyzing the source program. As it does so, it simulates the intended behavior of the source program by calling appropriate primitive subroutines from the library.

In this section, we explore the alternative strategy of *compilation*. A compiler for a given source language and machine translates a source program into an equivalent program (called the *object program*) written in the machine's native language. The compiler that we implement in this section translates programs written in JavaScript into sequences of instructions to be executed using the explicit-control evaluator machine's data paths.[33]

[32] This is a theoretical statement. We are not claiming that the evaluator's data paths are a particularly convenient or efficient set of data paths for a general-purpose computer. For example, they are not very good for implementing high-performance floating-point calculations or calculations that intensively manipulate bit vectors.

[33] Actually, the machine that runs compiled code can be simpler than the interpreter machine, because we

Compared with interpretation, compilation can provide a great increase in the efficiency of program execution, as we will explain below in the overview of the compiler. On the other hand, an interpreter provides a more powerful environment for interactive program development and debugging, because the source program being executed is available at run time to be examined and modified. In addition, because the entire library of primitives is present, new programs can be constructed and added to the system during debugging.

In view of the complementary advantages of compilation and interpretation, modern program-development environments pursue a mixed strategy. JavaScript interpreters are generally organized so that interpreted functions and compiled functions can call each other. This enables a programmer to compile those parts of a program that are assumed to be debugged, thus gaining the efficiency advantage of compilation, while retaining the interpretive mode of execution for those parts of the program that are in the flux of interactive development and debugging. In section 5.5.7, after we have implemented the compiler, we will show how to interface it with our interpreter to produce an integrated interpreter-compiler development system.

An overview of the compiler

Our compiler is much like our interpreter, both in its structure and in the function it performs. Accordingly, the mechanisms used by the compiler for analyzing expressions will be similar to those used by the interpreter. Moreover, to make it easy to interface compiled and interpreted code, we will design the compiler to generate code that obeys the same conventions of register usage as the interpreter: The environment will be kept in the `env` register, argument lists will be accumulated in `argl`, a function to be applied will be in `proc`, functions will return their answers in `val`, and the location to which a function should return will be kept in **`continue`**. In general, the compiler translates a source program into an object program that performs essentially the same register operations as would the interpreter in evaluating the same source program.

This description suggests a strategy for implementing a rudimentary compiler: We traverse the expression in the same way the interpreter does. When we encounter a register instruction that the interpreter would perform in evaluating the expression, we do not execute the instruction but instead accumulate it into a sequence. The resulting sequence of instructions will be the object code. Observe the efficiency advantage of compilation over interpretation. Each time the interpreter evaluates an expression—for example, `f(84, 96)`—it performs the work

won't use the `exp` and `unev` registers. The interpreter used these to hold pieces of unevaluated expressions. With the compiler, however, these expressions get built into the compiled code that the register machine will run. For the same reason, we don't need the machine operations that deal with expression syntax. But compiled code will use a few additional machine operations (to represent compiled function objects) that didn't appear in the explicit-control evaluator machine.

of classifying the expression (discovering that this is a function application) and testing for the end of the operand list (discovering that there are two operands). With a compiler, the expression is analyzed only once, when the instruction sequence is generated at compile time. The object code produced by the compiler contains only the instructions that evaluate the operator and the two operands, assemble the argument list, and apply the function (in `proc`) to the arguments (in `argl`).

This is the same kind of optimization we implemented in the analyzing evaluator of section 4.1.7. But there are further opportunities to gain efficiency in compiled code. As the interpreter runs, it follows a process that must be applicable to any expression in the language. In contrast, a given segment of compiled code is meant to execute some particular expression. This can make a big difference, for example in the use of the stack to save registers. When the interpreter evaluates an expression, it must be prepared for any contingency. Before evaluating a subexpression, the interpreter saves all registers that will be needed later, because the subexpression might require an arbitrary evaluation. A compiler, on the other hand, can exploit the structure of the particular expression it is processing to generate code that avoids unnecessary stack operations.

As a case in point, consider the combination `f(84, 96)`. Before the interpreter evaluates the operator of the combination, it prepares for this evaluation by saving the registers containing the operands and the environment, whose values will be needed later. The interpreter then evaluates the operator to obtain the result in `val`, restores the saved registers, and finally moves the result from `val` to `proc`. However, in the particular expression we are dealing with, the operator is the symbol `f`, whose evaluation is accomplished by the machine operation `lookup_variable_value`, which does not alter any registers. The compiler that we implement in this section will take advantage of this fact and generate code that evaluates the operator using the instruction

```
assign("proc", op("lookup_variable_value"), constant("f"), reg("env"));
```

This code not only avoids the unnecessary saves and restores but also assigns the value of the lookup directly to `proc`, whereas the interpreter would obtain the result in `val` and then move this to `proc`.

A compiler can also optimize access to the environment. Having analyzed the code, the compiler can in many cases know in which frame a particular variable will be located and access that frame directly, rather than performing the `lookup_variable_value` search. We will discuss how to implement such variable access in section 5.5.6. Until then, however, we will focus on the kind of register and stack optimizations described above. There are many other optimizations that can be performed by a compiler, such as coding primitive operations 'in line' instead of using a general `apply` mechanism (see exercise d.); but we will not emphasize these here. Our main goal in this section is to illustrate the compilation process in a simplified

(but still interesting) context.

5.5.1 Structure of the Compiler

In section 4.1.7 we modified our original metacircular interpreter to separate analysis from execution. We analyzed each expression to produce an execution function that took an environment as argument and performed the required operations. In our compiler, we will do essentially the same analysis. Instead of producing execution functions, however, we will generate sequences of instructions to be run by our register machine.

The function `compile` is the top-level dispatch in the compiler. It corresponds to the `eval` function of section 4.1.1, the `analyze` function of section 4.1.7, and the `eval_dispatch` entry point of the explicit-control-evaluator in section 5.4.1. The compiler, like the interpreters, uses the expression-syntax functions defined in section 4.1.2.[34] `Compile` performs a case analysis on the syntactic type of the expression to be compiled. For each type of expression, it dispatches to a specialized *code generator*:

```
function compile(exp, target, linkage) {
    return is_self-evaluating(exp)
        ? compile_self_evaluating(exp, target, linkage)
        : is_quoted(exp)
        ? compile_quoted(exp, target, linkage)
        : is_variable(exp)
        ? compile_variable(exp, target, linkage)
        : is_assignment(exp)
        ? compile_assignment(exp, target, linkage)
        : is_definition(exp)
        ? compile_definition(exp, target, linkage)
        : is_conditional_expression(exp)
        ? compile(cond_if(exp), target, linkage)
        : is_function_expression(exp)
        ? compile_function_expression(exp, target, linkage)
        : is_block(exp)
        ? compile_block(begin_actions(exp), target, linkage)
        : is_conditional_statement(exp)
        ? compile_conditional_statement(exp, target, linkage)
        : is_application(exp)
        ? compile_application(exp, target, linkage)
        : error(exp, "Unknown expression type - - COMPILE");
}
```

[34] Notice, however, that our compiler is a Scheme program, and the syntax functions that it uses to manipulate expressions are the actual Scheme functions used with the metacircular evaluator. For the explicit-control evaluator, in contrast, we assumed that equivalent syntax operations were available as operations for the register machine. (Of course, when we simulated the register machine in Scheme, we used the actual Scheme functions in our register machine simulation.)

Targets and linkages

`Compile` and the code generators that it calls take two arguments in addition to the expression to compile. There is a *target*, which specifies the register in which the compiled code is to return the value of the expression. There is also a *linkage descriptor*, which describes how the code resulting from the compilation of the expression should proceed when it has finished its execution. The linkage descriptor can require that the code do one of the following three things:

- continue at the next instruction in sequence (this is specified by the linkage descriptor `next`),
- return from the function being compiled (this is specified by the linkage descriptor **`return`**), or
- jump to a named entry point (this is specified by using the designated label as the linkage descriptor).

For example, compiling the expression `5` (which is self-evaluating) with a target of the `val` register and a linkage of `next` should produce the instruction

```
assign("val", constant(5));
```

Compiling the same expression with a linkage of **`return`** should produce the instructions

```
assign("val", constant(5));
      go_to(reg("continue"));
```

In the first case, execution will continue with the next instruction in the sequence. In the second case, we will return from a function call. In both cases, the value of the expression will be placed into the target `val` register.

Instruction sequences and stack usage

Each code generator returns an *instruction sequence* containing the object code it has generated for the expression. Code generation for a compound expression is accomplished by combining the output from simpler code generators for component expressions, just as evaluation of a compound expression is accomplished by evaluating the component expressions.

The simplest method for combining instruction sequences is a function called `append_instruction_sequences`. It takes as arguments any number of instruction sequences that are to be executed sequentially; it appends them and returns the combined sequence. That is, if seq_1 and seq_2 are sequences of instructions, then evaluating

```
append_instruction_sequences(seq1, seq2);
```

produces the sequence

```
seq1
seq2
```

Whenever registers might need to be saved, the compiler's code generators use `preserving`, which is a more subtle method for combining instruction sequences. `Preserving` takes three arguments: a set of registers and two instruction sequences that are to be executed sequentially. It appends the sequences in such a way that the contents of each register in the set is preserved over the execution of the first sequence, if this is needed for the execution of the second sequence. That is, if the first sequence modifies the register and the second sequence actually needs the register's original contents, then `preserving` wraps a `save` and a `restore` of the register around the first sequence before appending the sequences. Otherwise, `preserving` simply returns the appended instruction sequences. Thus, for example,

```
preserving(list(reg1, reg2), seq1, seq2);
```

produces one of the following four sequences of instructions, depending on how seq_1 and seq_2 use reg_1 and reg_2:

$\langle seq_1 \rangle$	`save(`$\langle reg_1 \rangle$`)`	`save(`$\langle reg_2 \rangle$`)`	`save(`$\langle reg_2 \rangle$`)`
$\langle seq_2 \rangle$	$\langle seq_1 \rangle$	$\langle seq_1 \rangle$	`save(`$\langle reg_1 \rangle$`)`
	`restore(`$\langle reg_1 \rangle$`)`	`restore(`$\langle reg_2 \rangle$`)`	$\langle seq_1 \rangle$
	$\langle seq_2 \rangle$	$\langle seq_2 \rangle$	`restore(`$\langle reg_1 \rangle$`)`
			`restore(`$\langle reg_2 \rangle$`)`
			$\langle seq_2 \rangle$

By using `preserving` to combine instruction sequences the compiler avoids unnecessary stack operations. This also isolates the details of whether or not to generate `save` and `restore` instructions within the `preserving` function, separating them from the concerns that arise in writing each of the individual code generators. In fact no `save` or `restore` instructions are explicitly produced by the code generators.

In principle, we could represent an instruction sequence simply as a list of instructions. `Append_instruction_s` could then combine instruction sequences by performing an ordinary list `append`. However, `preserving` would then be a complex operation, because it would have to analyze each instruction sequence to determine how the sequence uses its registers. `Preserving` would be inefficient as well as complex, because it would have to analyze each of its instruction sequence arguments, even though these sequences might themselves have been constructed by calls to `preserving`, in which case their parts would have already been analyzed. To avoid such repetitious analysis we will associate with each instruction sequence some information about its register use. When we construct a basic instruction sequence we will provide this information explicitly, and the functions that combine instruction sequences will derive register-use information for the combined sequence from the information associated with the component sequences.

An instruction sequence will contain three pieces of information:

- the set of registers that must be initialized before the instructions in the sequence are executed (these registers are said to be *needed* by the sequence),
- the set of registers whose values are modified by the instructions in the sequence, and
- the actual instructions (also called *statements*) in the sequence.

We will represent an instruction sequence as a list of its three parts. The constructor for instruction sequences is thus

```
function make_instruction_sequence(needs, modifies, statements) {
    return list(needs, modifies,statements);
}
```

For example, the two-instruction sequence that looks up the value of the variable `x` in the current environment, assigns the result to `val`, and then returns, requires registers `env` and **`continue`** to have been initialized, and modifies register `val`. This sequence would therefore be constructed as

```
make_instruction_sequence(list("env", "continue"),
                          list("val"),
                          list(assign("val", op("lookup_variable_value"), constant("
                               go_to(reg("continue"))));
```

We sometimes need to construct an instruction sequence with no statements:

```
function empty_instruction_sequence() {
    return make_instruction_sequence(list(), list(), list());
}
```

The functions for combining instruction sequences are shown in section 5.5.4.

Exercise 5.31

In evaluating a function application, the explicit-control evaluator always saves and restores the `env` register around the evaluation of the operator, saves and restores `env` around the evaluation of each operand (except the final one), saves and restores `argl` around the evaluation of each operand, and saves and restores `proc` around the evaluation of the operand sequence. For each of the following combinations, say which of these `save` and `restore` operations are superfluous and thus could be eliminated by the compiler's `preserving` mechanism:

Exercise 5.32

Using the `preserving` mechanism, the compiler will avoid saving and restoring `env` around the evaluation of the operator of a combination in the case where the operator is a symbol. We could also build such optimizations into the evaluator. Indeed, the explicit-control evaluator of section 5.4 already performs a similar optimization, by treating combinations with no operands as a special case.

a. Extend the explicit-control evaluator to recognize as a separate class of expressions combinations whose operator is a symbol, and to take advantage of this fact in evaluating such expressions.

b. Alyssa P. Hacker suggests that by extending the evaluator to recognize more and more special cases we could incorporate all the compiler's optimizations, and that this would eliminate the advantage of compilation altogether. What do you think of this idea?

5.5.2 Compiling Expressions

In this section and the next we implement the code generators to which the `compile` function dispatches.

Compiling linkage code

In general, the output of each code generator will end with instructions—generated by the function `compile_linkage`—that implement the required linkage. If the linkage is **`return`** then we must generate the instruction `go_to(reg("continue"))`. This needs the **`continue`** register and does not modify any registers. If the linkage is `next`, then we needn't include any additional instructions. Otherwise, the linkage is a label, and we generate a `go_to` to that label, an instruction that does not need or modify any registers.[35]

For example, if the value of `linkage` is the string `"branch25"`, then the expression `go_to(label(linkage))` evaluates to the list `go_to(label("branch25"))`. Similarly, if the value of `x` is the list `list("a", "b", "c")`, then `list(1, 2, head(x))` evaluates to the list `list(1, 2, "a")`.

```
function compile_linkage(linkage) {
    return linkage === "return"
        ? make_instruction_sequence(list("continue"), list(), list(go_to(reg("contin
        : linkage === "next"
        ? empty_instruction_sequence()
        : make_instruction_sequence(list(), list(), list(go_to(label(linkage))));
}
```

[35] This function uses a feature of JavaScript called *backquote* (or *quasiquote*) that is handy for constructing lists. Preceding a list with a backquote symbol is much like quoting it, except that anything in the list that is flagged with a comma is evaluated.

The linkage code is appended to an instruction sequence by preserving the **continue** register, since a **return** linkage will require the **continue** register: If the given instruction sequence modifies **continue** and the linkage code needs it, **continue** will be saved and restored.

```
function end_with_linkage(linkage, instruction_sequence) {
    return preserving(list("continue"), instruction_sequence, compile_linkage(linkag
}
```

Compiling simple expressions

The code generators for self-evaluating expressions, quotations, and variables construct instruction sequences that assign the required value to the target register and then proceed as specified by the linkage descriptor.

```
function compile_self_evaluating(exp, target, linkage) {
        return end_with_linkage(
            linkage,
            make_instruction_sequence(
                list(),
                list(target),
                list(assign(target, constant(exp)))));
    }


    function compile_quoted(exp, target, linkage) {
        return end_with_linkage(
            linkage,
            make_instruction_sequence(
                list(),
                list(target),
                list(assign(target, constant(stringify(exp))))));
    }


    function compile_variable(exp, target, linkage) {
        return end_with_linkage(
            linkage,
            make_instruction_sequence(
                list("env"),
                list(target),
                list(assign(target, op("lookup_variable_value"), constant(exp), reg(
    }
```

All these assignment instructions modify the target register, and the one that looks up a variable needs the env register.

Assignments and definitions are handled much as they are in the interpreter. We recursively

generate code that computes the value to be assigned to the variable, and append to it a two-instruction sequence that actually sets or defines the variable and assigns the value of the whole expression (the symbol ok) to the target register. The recursive compilation has target val and linkage next so that the code will put its result into val and continue with the code that is appended after it. The appending is done preserving env, since the environment is needed for setting or defining the variable and the code for the variable value could be the compilation of a complex expression that might modify the registers in arbitrary ways.

```
function compile_assignment(exp, target, linkage) {
    const variable = assignment_variable(exp);
    const get_value_code = compile(assignment_value(exp), "val", "next");

    return end_with_linkage(
        linkage,
        preserving(
            "env",
            get_value_code,
            make_instruction_sequence(
                list("env", "val"),
                list(target),
                list(
                    perform(
                        op("set_variable_value"),
                        constant(variable),
                        reg("val"),
                        reg("env")),
                    assign(target, constant(ok))))); /// FIXME: ok??=
}

function compile_definition(exp, target, linkage) {
    const variable = definition_variable(exp);
    const get_value_code = compile(definition_value(exp), "val", "next");

    return end_with_linkage(
        linkage,
        preserving(
            "env",
            get_value_code,
            make_instruction_sequence()
                list("env", "val"),
                list(target),
                list(
                    perform(
                        op("define_variable"),
                        constant(variable),
                        reg("val"),
                        reg("env")),
```

```
                    assign(target, constant(ok))))); /// FIXME: ok??=
}
```

The appended two-instruction sequence requires `env` and `val` and modifies the target. Note that although we preserve `env` for this sequence, we do not preserve `val`, because the `get_value_code` is designed to explicitly place its result in `val` for use by this sequence. (In fact, if we did preserve `val`, we would have a bug, because this would cause the previous contents of `val` to be restored right after the `get_value_code` is run.)

Compiling conditional expressions

The code for an **`if`** expression compiled with a given target and linkage has the form

```
⟨compilation of predicate, target val, linkage next ⟩
      test(op("is_false"), reg("val")),
          branch(label("false_branch")),
      "true_branch",
      ⟨compilation of consequent with given target and given linkage or after_if ⟩
      "false_branch",
      ⟨compilation of alternative with given target and linkage⟩
      "after_if",
```

To generate this code, we compile the predicate, consequent, and alternative, and combine the resulting code with instructions to test the predicate result and with newly generated labels to mark the true and false branches and the end of the conditional.[36] In this arrangement of code, we must branch around the true branch if the test is false. The only slight complication is in how the linkage for the true branch should be handled. If the linkage for the conditional is **`return`** or a label, then the true and false branches will both use this same linkage. If the linkage is `next`, the true branch ends with a jump around the code for the false branch to the

[36] We can't just use the labels `true_branch`, `false_branch`, and `after_if` as shown above, because there might be more than one **`if`** in the program. The compiler uses the function `make_label` to generate labels. `Make_label` takes a symbol as argument and returns a new symbol that begins with the given symbol. For example, successive calls to `make_label("a")` would return `a1`, `a2`, and so on. `Make_label` can be implemented similarly to the generation of unique variable names in the query language, as follows:

```
let label_counter = 0;

        function new_label_number() {
            label_counter = label_counter + 1;
            return label-counter;
        }



        function make_label(name) {
            return name + stringify(new_label_number());
        }
```

label at the end of the conditional.

```
function compile_if(exp, target, linkage) {
    let t_branch = make_label("true_branch");
    let f_branch = make_label("false_branch");
    let after_if = make_label("after_if");
    let consequent_linkage = linkage === "next" ? after_if : linkage;
    let p_code = compile(if_predicate(exp), "val", "next");
    let c_code = compile(if_consequent(exp), target, consequent_linkage);
    let a_code = compile(if_alternative(exp), target, linkage);

    return preserving(list("env", "continue"),
                      p_code,
                      append_instruction_sequences(
                          make_instruction_sequence(list("val"),
                                                    list(),
                                                    list(
                                                        test(op("is_false"), reg("va
                                                        branch(label(f_branch)))),
                          parallel_instruction_sequences(
                              append_instruction_sequences(t_branch, c_code),
                              append_instruction_sequences(f_branch, a_code)),
                          after_if));
}
```

Env is preserved around the predicate code because it could be needed by the true and false branches, and **continue** is preserved because it could be needed by the linkage code in those branches. The code for the true and false branches (which are not executed sequentially) is appended using a special combiner parallel_instruction_sequences described in section 5.5.4.

FIXME: Note that cond is a derived expression, so all that the compiler needs to do handle it is to apply the cond->**if** transformer (from section 4.1.2) and compile the resulting **if** expression.

Compiling sequences

The compilation of sequences (from function bodies or explicit begin expressions) parallels their evaluation. Each expression of the sequence is compiled—the last expression with the linkage specified for the sequence, and the other expressions with linkage next (to execute the rest of the sequence). The instruction sequences for the individual expressions are appended to form a single instruction sequence, such that env (needed for the rest of the sequence) and **continue** (possibly needed for the linkage at the end of the sequence) are preserved.

```
function compile_sequence(seq, target, linkage) {
    return is_last_exp(seq)
        ? compile(first_exp(seq), target, linkage)
        : preserving(
            list("env", "continue"),
```

```
            compile(first_exp("seq"), target, "next"),
            compile_sequence(rest_exps(seq), target, linkage));
}
```

Compiling `lambda` expressions

Function definition expressions expressions construct functions. The object code for a function definition expression must have the form

```
⟨construct procedure object and assign it to target register⟩
⟨linkage⟩
```

When we compile the function definition expression, we also generate the code for the function body. Although the body won't be executed at the time of function construction, it is convenient to insert it into the object code right after the code for the function definition. If the linkage for the function definition expression is a label or **`return`**, this is fine. But if the linkage is `next`, we will need to skip around the code for the function body by using a linkage that jumps to a label that is inserted after the body. The object code thus has the form

```
⟨construct procedure object and assign it to target register⟩
⟨code for given linkage⟩ or go_to(label("after_lambda"))
⟨compilation of procedure body⟩
"after_lambda",
```

`Compile_function_expression` generates the code for constructing the function object followed by the code for the function body. The function object will be constructed at run time by combining the current environment (the environment at the point of definition) with the entry point to the compiled function body (a newly generated label).[37]

[37] We need machine operations to implement a data structure for representing compiled functions, analogous to the structure for compound functions described in section 4.1.3:

```
function make_compiled_procedure(entry, env) {
            return list("compiled_procedure", entry, env);
        }


        function is_compiled_procedure(proc) {
            return is_tagged_list(proc, "compiled_procedure");
        }


        function compiled_procedure_entry(c_proc) {
            return head(tail(c_proc));
        }


        function compiled_procedure_env(c_proc) {
            return head(tail(tail(c_proc)));
        }
```

```
function compile_function_expression(exp, target, linkage) {
    let proc_entry = make_label("entry");
    let after_fexp = make_label("after_lambda"); /// FIXME: lambda
    let lambda_linkage = linkage === "next" ? after_lambda : linkage;

    return append_instruction_sequences(
        tack_on_instruction_sequence(
            end_with_linkage(
                lambda_linkage,
                make_instruction_sequence(
                    list("env"),
                    list(target),
                    list(assign(target, op("make_compiled_procedure"), label(proc_en
            compile_function_expression_body(exp, proc_entry)),
            after_lambda));
}
```

`Compile_function_expression` uses the special combiner `tack_on_instruction_sequence` (section 5.5.4) rather than `append_instruction_sequences` to append the function body to the `lambda` expression code, because the body is not part of the sequence of instructions that will be executed when the combined sequence is entered; rather, it is in the sequence only because that was a convenient place to put it.

`Compile_lambda_body` constructs the code for the body of the function. This code begins with a label for the entry point. Next come instructions that will cause the run-time evaluation environment to switch to the correct environment for evaluating the function body—namely, the definition environment of the function, extended to include the bindings of the formal parameters to the arguments with which the function is called. After this comes the code for the sequence of expressions that makes up the function body. The sequence is compiled with linkage **`return`** and target `val` so that it will end by returning from the function with the function result in `val`.

```
function compile_lambda_body(exp, proc_entry) {
    let formals  = lambda_parameters(exp);
    return append_instruction_sequences(
        make_instruction_sequence(
            list("env", "proc", "argl"),
            list("env"),
            list(proc_entry,
                 assign("env", op("compiled_procedure_env"), reg("proc")),
                 assign("env", op("extend_environment"), constant(formals), reg("arg
        compiled_sequence(lambda_body(exp), "val", "return"));
}
```

5.5.3 Compiling Combinations

The essence of the compilation process is the compilation of function applications. The code for a combination compiled with a given target and linkage has the form

```
⟨compilation of operator, target proc, linkage next⟩
⟨evaluate operands and construct argument list in argl⟩
⟨compilation of procedure call with given target and linkage⟩
```

The registers `env`, `proc`, and `argl` may have to be saved and restored during evaluation of the operator and operands. Note that this is the only place in the compiler where a target other than `val` is specified.

The required code is generated by `compile_application`. This recursively compiles the operator, to produce code that puts the function to be applied into `proc`, and compiles the operands, to produce code that evaluates the individual operands of the application. The instruction sequences for the operands are combined (by `construct_arglist`) with code that constructs the list of arguments in `argl`, and the resulting argument-list code is combined with the function code and the code that performs the function call (produced by `compile_procedure_call`). In appending the code sequences, the `env` register must be preserved around the evaluation of the operator (since evaluating the operator might modify `env`, which will be needed to evaluate the operands), and the `proc` register must be preserved around the construction of the argument list (since evaluating the operands might modify `proc`, which will be needed for the actual function application). `Continue` must also be preserved throughout, since it is needed for the linkage in the function call.

```
function compile_application(exp, target, linkage) {
    const proc_code = compile(operator(exp), "proc", "next");
    const operand_codes = map(operand => compile(operand, "val", "next"), operands(e

    return preserving(list("env", "continue"),
                      preserving(list("proc", "continue"),
                                 construct_arglist(operand_codes),
                                 compile_procedure_call(target, linkage)));
}
```

The code to construct the argument list will evaluate each operand into `val` and then `pair` that value onto the argument list being accumulated in `argl`. Since we `pair` the arguments onto `argl` in sequence, we must start with the last argument and end with the first, so that the arguments will appear in order from first to last in the resulting list. Rather than waste an instruction by initializing `argl` to the empty list to set up for this sequence of evaluations, we make the first code sequence construct the initial `argl`. The general form of the argument-list construction is thus as follows:

```
⟨compilation of last operand, targeted to val⟩
```

```
assign("argl", list(op("list"), reg("val")));
⟨compilation of next operand, targeted to val⟩
assign("argl", list(op("pair"), reg("val"), reg("argl")));
...
⟨compilation of first operand, targeted to val⟩
assign("argl", list(op("pair"), reg("val"), reg("argl")));
```

`Argl` must be preserved around each operand evaluation except the first (so that arguments accumulated so far won't be lost), and `env` must be preserved around each operand evaluation except the last (for use by subsequent operand evaluations).

Compiling this argument code is a bit tricky, because of the special treatment of the first operand to be evaluated and the need to preserve `argl` and `env` in different places. The `construct_arglist` function takes as arguments the code that evaluates the individual operands. If there are no operands at all, it simply emits the instruction

```
assign(argl, list(constant("null")));
```

Otherwise, `construct_arglist` creates code that initializes `argl` with the last argument, and appends code that evaluates the rest of the arguments and adjoins them to `argl` in succession. In order to process the arguments from last to first, we must reverse the list of operand code sequences from the order supplied by `compile_application`.

```
function construct_arglist(operand_codes) {
    const operand_codes = reverse(operand_codes);

    if (is_null(operand_codes)) {
        return make_instruction_sequence(
            null,
            list("argl"),
            list(assign("argl", list(constant("null")))));
    } else {
        const code_to_get_last_arg =
              append_instruction_sequences(
                  head(operand_codes),
                  make_instruction_sequence(
                      list("val"),
                      list("argl"),
                      list(assign("argl", list(op("list"), reg("val"))))));

        return is_null(tail(operand_codes))
            ? code_to_get_last_arg
            : preserving(
                list("env"),
                code_to_get_last_arg,
                code_to_get_rest_args(tail(operand_codes)));
    }
}
```

```
function code_to_get_rest_args(operand_codes) {
    const code_for_next_arg = preserving(
        list("argl"),
        head(operand_codes),
        make_instruction_sequence(
            list("val", "argl"),
            list("argl"),
            list(assign("argl", list(op("cons"), reg("val"), reg("argl"))))));

    return is_null(tail(operand_codes))
        ? code_for_next_arg
        : preserving(list("env"),
                     code_for_next_arg,
                     code_to_get_rest_args(tail(operand_codes)));
}
```

Applying functions

After evaluating the elements of a combination, the compiled code must apply the function in `proc` to the arguments in `argl`. The code performs essentially the same dispatch as the `apply` function in the meta-circular evaluator of section 4.1.1 or the `apply-dispatch` entry point in the explicit-control evaluator of section 5.4.1. It checks whether the function to be applied is a primitive function or a compiled function. For a primitive function, it uses `apply_primitive_function`; we will see shortly how it handles compiled functions. The function-application code has the following form:

```
test(op("primitive-procedure"), reg("proc")),
  branch(label("primitive_branch")),
"compiled-branch",
  ⟨code to apply compiled procedure with given target and appropriate linkage⟩
"primitive-branch",
  assign(⟨target⟩,
         list(op("apply_primitive_procedure"),
         reg("proc"),
         reg("argl")))
⟨linkage⟩
"after_call",
```

Observe that the compiled branch must skip around the primitive branch. Therefore, if the linkage for the original function call was `next`, the compound branch must use a linkage that jumps to a label that is inserted after the primitive branch. (This is similar to the linkage used for the true branch in `compile_if`.)

```
function compile_procedure_call(target, linkage) {
    const primitive_branch = make_label("primitive-branch");
```

```
    const compiled_branch = make_label("compiled-branch");
    const after_call = make_label("after-call");

    const compiled_linkage = linkage === "next" ? after_call : linkage;

    return append_instruction_sequences(
        make_instruction_sequence(
            list("proc"),
            list(),
            list(test(op("primitive_procedure"), reg("proc"), branch(label(primitive
        parallel_instruction_sequences(
            append_instruction_sequences(
                compiled_branch,
                compile_proc_appl(target, compiled_linkage)),
            append_instruction_sequences(
                primitive_branch,
                end_with_linkage(
                    linkage,
                    make_instruction_sequence(list("proc", "argl"),
                                              list(target),
                                              assign(target, list(op("apply_primitiv
        "after_call");
}
```

The primitive and compound branches, like the true and false branches in `compile_if`, are appended using `parallel_instruction_sequences` rather than the ordinary `append_instruction_sequences`, because they will not be executed sequentially.

Applying compiled functions

The code that handles function application is the most subtle part of the compiler, even though the instruction sequences it generates are very short. A compiled function (as constructed by `compile_function_definition_expression`) has an entry point, which is a label that designates where the code for the function starts. The code at this entry point computes a result in `val` and returns by executing the instruction `go_to(reg("continue"))`. Thus, we might expect the code for a compiled-function application (to be generated by `compile_proc_appl`) with a given target and linkage to look like this if the linkage is a label

```
assign("continue", list(label("proc_return")),)
      assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
      go_to(reg("val")),
      "proc_return",

      assign(⟨target⟩, list(reg("val"))), // included if target is not val

      go_to(label(⟨linkage⟩)), // linkage code
```

or like this if the linkage is **return**.

```
save("continue"),
      assign("continue", list(label("proc_return"))),
      assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
      go_to(reg("val")),
      "proc_return",
      assign(⟨target⟩, list(reg("val"))), // included if target is not val
      restore("continue"),
      go_to(reg("continue")), // linkage code
```

This code sets up **continue** so that the function will return to a label `proc_return` and jumps to the function's entry point. The code at `proc_return` transfers the function's result from `val` to the target register (if necessary) and then jumps to the location specified by the linkage. (The linkage is always **return** or a label, because `compile_procedure_call` replaces a `next` linkage for the compound-procedure branch by an `after_call` label.)

In fact, if the target is not `val`, that is exactly the code our compiler will generate.[38] Usually, however, the target is `val` (the only time the compiler specifies a different register is when targeting the evaluation of an operator to `proc`), so the function result is put directly into the target register and there is no need to return to a special location that copies it. Instead, we simplify the code by setting up **continue** so that the function will 'return' directly to the place specified by the caller's linkage:

```
⟨set up⟩ continue ⟨for linkage⟩
      assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
      go_to(reg("val")),
```

If the linkage is a label, we set up **continue** so that the function will return to that label. (That is, the `go_to(reg("continue"))` the function ends with becomes equivalent to the `go_to(label(`⟨*linkage*⟩`))` at `proc_return` above.)

```
assign("continue", list(label(⟨linkage⟩))),
      assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
      go_to(reg("val")),
```

If the linkage is **return**, we don't need to set up **continue** at all: It already holds the desired location. (That is, the `go_to(reg("continue"))` the function ends with goes directly to the place where the `go_to(reg("continue"))` at `proc_return` would have gone.)

```
assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
      go_to(reg("val")),
```

With this implementation of the **return** linkage, the compiler generates tail-recursive code. Calling a function as the final step in a function body does a direct transfer, without saving any information on the stack.

[38] Actually, we signal an error when the target is not `val` and the linkage is **return**, since the only place we request **return** linkages is in compiling functions, and our convention is that functions return their values in `val`.

Suppose instead that we had handled the case of a function call with a linkage of **return** and a target of val as shown above for a non-val target. This would destroy tail recursion. Our system would still give the same value for any expression. But each time we called a function, we would save **continue** and return after the call to undo the (useless) save. These extra saves would accumulate during a nest of function calls.[39]

Compile_proc_appl generates the above function-application code by considering four cases, depending on whether the target for the call is val and whether the linkage is **return**. Observe that the instruction sequences are declared to modify all the registers, since executing the function body can change the registers in arbitrary ways.[40] Also note that the code sequence for the case with target val and linkage **return** is declared to need **continue**: Even though **continue** is not explicitly used in the two-instruction sequence, we must be sure that **continue** will have the correct value when we enter the compiled function.

```
function compile_proc_appl(target, linkage) {
    if (target === "val" && linkage !== "return") {
        return make_instruction_sequence(
            list("proc"),
            all_regs,
            list(
                assign("continue", list(label(linkage))),
                assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
                go_to(reg("val"))));
    } else if (target !== "val" && linkage !== "return") {
        const proc_return = make_label("proc_return");

        return make_instruction_sequence(
            list("proc"),
            all_regs,
            list(
                assign("continue", list(label(proc_return))),
                assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
                go_to(reg("val")),
                proc_return,
```

[39] Making a compiler generate tail-recursive code might seem like a straightforward idea. But most compilers for common languages, including C and Pascal, do not do this, and therefore these languages cannot represent iterative processes in terms of function call alone. The difficulty with tail recursion in these languages is that their implementations use the stack to store function arguments and local variables as well as return addresses. The Scheme implementations described in this book store arguments and variables in memory to be garbage-collected. The reason for using the stack for variables and arguments is that it avoids the need for garbage collection in languages that would not otherwise require it, and is generally believed to be more efficient. Sophisticated JavaScript compilers can, in fact, use the stack for arguments without destroying tail recursion. (See Hanson 1990 for a description.) There is also some debate about whether stack allocation is actually more efficient than garbage collection in the first place, but the details seem to hinge on fine points of computer architecture. (See Appel 1987 and Miller and Rozas 1994 for opposing views on this issue.)

[40] The variable all_regs is bound to the list of names of all the registers:

```
const all_regs = list("env", "proc", "val", "argl", "continue");
```

```
                assign(target, list(reg("val"))),
                go_to(label(linkage))));
    } else if (target === "val" && linkage === "return") {
        return make_instruction_sequence(
            list("proc", "continue"),
            all_regs,
            list(
                assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
                go_to(reg("val"))));
    } else if (target !== "val" && linkage === "return") {
        error(target, "return linkage, target not val - - COMPILE");
    }
}
```

5.5.4 Combining Instruction Sequences

This section describes the details on how instruction sequences are represented and combined. Recall from section 5.5.1 that an instruction sequence is represented as a list of the registers needed, the registers modified, and the actual instructions. We will also consider a label (symbol) to be a degenerate case of an instruction sequence, which doesn't need or modify any registers. So to determine the registers needed and modified by instruction sequences we use the selectors

```
function registers_needed(s) {
     return is_symbol(s) ? null : head(s);
}


function registers_modified(s) {
     return is_symbol(s) ? null : head(tail(s));
}


function statements(s) {
     return is_symbol(s) ? list(s) : head(tail(tail(s));
}
```

and to determine whether a given sequence needs or modifies a given register we use the predicates

```
function needs_register(seq, reg) {
    return member(reg(registers_needed(seq))); /// FIXME: see https://github.com/sou
}


function modifies_register(seq, reg) {
    return member(reg(registers_modified(seq)));
}
```

In terms of these predicates and selectors, we can implement the various instruction sequence combiners used throughout the compiler.

The basic combiner is append_instruction_sequences. This takes as arguments an arbitrary number of instruction sequences that are to be executed sequentially and returns an instruction sequence whose statements are the statements of all the sequences appended together. The subtle point is to determine the registers that are needed and modified by the resulting sequence. It modifies those registers that are modified by any of the sequences; it needs those registers that must be initialized before the first sequence can be run (the registers needed by the first sequence), together with those registers needed by any of the other sequences that are not initialized (modified) by sequences preceding it.

The sequences are appended two at a time by append_2_sequences. This takes two instruction sequences seq1 and seq2 and returns the instruction sequence whose statements are the statements of seq1 followed by the statements of seq2, whose modified registers are those registers that are modified by either seq1 or seq2, and whose needed registers are the registers needed by seq1 together with those registers needed by seq2 that are not modified by seq1. (In terms of set operations, the new set of needed registers is the union of the set of registers needed by seq1 with the set difference of the registers needed by seq2 and the registers modified by seq1.) Thus, append_instruction_sequences is implemented as follows:

```
function append_instruction_sequences( . seqs) {
    function append_2_sequences(seq1, seq2) {
        return make_instruction_sequence(
            list_union(registers_needed(seq1),
                       list_difference(registers_needed(seq2)
                                       registers_modified(seq1))),
            list_union(registers_modified(seq1),
                       registers_modified(seq2)),
            append(statements(seq1), statements(seq2)));
    }

    function append_seq_list(seqs) {
        return is_null(seqs)
            ? empty_instruction_sequence
            : append_2_sequences(
                head(seqs),
                append_seq_list(tail(seqs)));
    }

    return append_seq_list(seqs);
}
```

This function uses some simple operations for manipulating sets represented as lists, similar to the (unordered) set representation described in section **??**:

```
function list_union(s1, s2) {
    return is_null(s1)
        ? s2
        : member(head(s1), s2)
        ? list_union(tail(s1), s2)
        : pair(head(s1), list_union(tail(s1), s2));
}


function list_difference(s1, s2) {
    return is_null(s1)
        ? null
        : member(head(s1), s2)
        ? list_difference(tail(s1), s2)
        : pair(head(s1), list_difference(tail(s1), s2));
}
```

`Preserving`, the second major instruction sequence combiner, takes a list of registers `regs` and two instruction sequences `seq1` and `seq2` that are to be executed sequentially. It returns an instruction sequence whose statements are the statements of `seq1` followed by the statements of `seq2`, with appropriate `save` and `restore` instructions around `seq1` to protect the registers in `regs` that are modified by `seq1` but needed by `seq2`. To accomplish this, `preserving` first creates a sequence that has the required `save`s followed by the statements of `seq1` followed by the required `restore`s. This sequence needs the registers being saved and restored in addition to the registers needed by `seq1`, and modifies the registers modified by `seq1` except for the ones being saved and restored. This augmented sequence and `seq2` are then appended in the usual way. The following function implements this strategy recursively, walking down the list of registers to be preserved:[41]

```
function preserving(regs, seq1, seq2) {
    if (is_null(regs)) {
        return append_instruction_sequences(seq1, seq2)
    } else {
        const first_reg = head(regs);

        if (need_register(seq2, first_reg) && modifies_register(seq1, first_reg)) {
            return preserving(
                tail(regs),
                make_instruction_sequence(
                    list_union(list(first_reg),
                               registers_needed(seq1)),
                    list_difference(registers_modified(seq1),
                                    list(first_reg)),
```

[41] Note that `preserving` calls `append` with three arguments. Though the definition of `append` shown in this book accepts only two arguments, Scheme standardly provides an `append` function that takes an arbitrary number of arguments.

```
                    append(list(save(first_reg)),
                           append(statements(seq1),
                                  list(restore(first_reg)))))
                seq2);
        } else {
            return preserving(tail(regs), seq1, seq2);
        }
    }
}
```

Another sequence combiner, `tack_on_instruction_sequence`, is used by `compile_lambda` to append a function body to another sequence. Because the function body is not 'in line' to be executed as part of the combined sequence, its register use has no impact on the register use of the sequence in which it is embedded. We thus ignore the function body's sets of needed and modified registers when we tack it onto the other sequence.

```
function tack_on_instruction_sequence(seq, body_seq) {
    return make_instruction_sequence(
        registers_needed(seq),
        registers_modified(seq),
        append(statements(seq), statements(body_seq)));
}
```

`Compile_if` and `compile_procedure_call` use a special combiner called `parallel-instruction-sequences` to append the two alternative branches that follow a test. The two branches will never be executed sequentially; for any particular evaluation of the test, one branch or the other will be entered. Because of this, the registers needed by the second branch are still needed by the combined sequence, even if these are modified by the first branch.

```
function parallel_instruction_sequences(seq1, seq2) {
    return make_instruction_sequence(
        list_union(
            registers_needed(seq1),
            registers_needed(seq2)),
        list_union(
            registers_modified(seq1),
            registers_modified(seq2)),
        append(
            statements(seq1),
            statements(seq2)));
}
```

5.5.5 An Example of Compiled Code

Note: this section is a work in progress!

Now that we have seen all the elements of the compiler, let us examine an example of compiled code to see how things fit together. We will compile the definition of a recursive `factorial` function by calling `compile`:

```
compile(parse("                \
function factorial(n) {        \
    return n === 1             \
        ? 1"                   \
        : n * factorial(n - 1);\
}                              "),
       "val",
       "next");
```

We have specified that the value of the `define` expression should be placed in the `val` register. We don't care what the compiled code does after executing the `define`, so our choice of `next` as the linkage descriptor is arbitrary.

`Compile` determines that the expression is a definition, so it calls `compile_definition` to compile code to compute the value to be assigned (targeted to `val`), followed by code to install the definition, followed by code to put the value of the `define` (which is the symbol `ok`) into the target register, followed finally by the linkage code. `Env` is preserved around the computation of the value, because it is needed in order to install the definition. Because the linkage is `next`, there is no linkage code in this case. The skeleton of the compiled code is thus

```
⟨save env if modified by code to compute value⟩
⟨compilation of definition value, target val,linkage next⟩
⟨restore env if saved above⟩
perform(
    op("define_variable"),
    constant("factorial"),
    reg("val"),
    reg("env"));
assign("val", constant(ok));
```

The expression that is to be compiled to produce the value for the variable `factorial` is a `lambda` expression whose value is the function that computes factorials. `Compile` handles this by calling `compile_lambda`, which compiles the function body, labels it as a new entry point, and generates the instruction that will combine the function body at the new entry point with the run-time environment and assign the result to `val`. The sequence then skips around the compiled function code, which is inserted at this point. The function code itself begins by extending the function's definition environment by a frame that binds the formal parameter `n`

to the function argument. Then comes the actual function body. Since this code for the value of the variable doesn't modify the env register, the optional save and restore shown above aren't generated. (The function code at entry2 isn't executed at this point, so its use of env is irrelevant.) Therefore, the skeleton for the compiled code becomes

```
assign("val", list(op("make_compiled_function"),
                   label(entry2),
                   reg("env"))),
  go_to(label("after_lambda1")),
"entry2",
  assign("env", list(op("compiled_function_env"), reg("fun"))),
  assign("env", list(op("extend_environment"),
                   constant(n),
                   reg("argl"),
                   reg("env"))),
  ⟨compilation of procedure body⟩
"after_lambda1",
  perform(op("define-variable"),
          constant("factorial"),
          reg("val"),
          reg("env")),
  assign("val", constant(ok)),
```

A function body is always compiled (by compile_lambda_body) as a sequence with target val and linkage **return**. The sequence in this case consists of a single **if** expression:

```
n === 1
    ? 1
    : n * factorial(n - 1)
```

Compile_if generates code that first computes the predicate (targeted to val), then checks the result and branches around the true branch if the predicate is false. Env and **continue** are preserved around the predicate code, since they may be needed for the rest of the **if** expression. Since the **if** expression is the final expression (and only expression) in the sequence making up the function body, its target is val and its linkage is **return**, so the true and false branches are both compiled with target val and linkage **return**. (That is, the value of the conditional, which is the value computed by either of its branches, is the value of the function.)

```
⟨save continue, env if modified by predicate and needed by branches⟩
  ⟨compilation of predicate, target val,linkage next⟩
  ⟨restore continue, env if saved above⟩
  test(op("false"), reg("val")),
  branch(label("false_branch4")),
"true_branch5",
  ⟨compilation of true branch, target val,linkage return⟩
"false_branch4",
  ⟨compilation of false branch, target val,linkage return⟩
"after_if3",
```

The predicate `n === 1` is a function call. This looks up the operator (the symbol =) and places this value in `fun`. It then assembles the arguments `1` and the value of `n` into `argl`. Then it tests whether `fun` contains a primitive or a compound function, and dispatches to a primitive branch or a compound branch accordingly. Both branches resume at the `after_call` label. The requirements to preserve registers around the evaluation of the operator and operands don't result in any saving of registers, because in this case those evaluations don't modify the registers in question.

```
assign("fun", list(op("lookup_variable_value"), constant("="), reg("env"))),
assign("val", constant(1)),
assign("argl", list(op("list"), reg("val"))),
assign("val", list(op("lookup_variable_value"), constant(n), reg("env"))),
assign("argl", list(op("cons"), reg("val"), reg("argl"))),
test(op("primitive_function"), reg("fun")),
branch(label("primitive_branch17")),
"compiled_branch16",
assign("continue", list(label("after_call15"))),
assign("val", list(op("compiled_function_entry"), reg("fun"))),
go_to(reg("val")),
"primitive_branch17",
assign("val", list(op("apply_primitive_function"), reg("fun"), reg("argl"))),
"after_call15",
```

The true branch, which is the constant 1, compiles (with target `val` and linkage **`return`**) to

```
assign("val", constant(1)),
go_to(reg("continue")),
```

The code for the false branch is another a function call, where the function is the value of the symbol `*`, and the arguments are `n` and the result of another function call (a call to `factorial`). Each of these calls sets up `fun` and `argl` and its own primitive and compound branches. Figure **??** shows the complete compilation of the definition of the `factorial` function. Notice that the possible `save` and `restore` of **`continue`** and `env` around the predicate, shown above, are in fact generated, because these registers are modified by the function call in the predicate and needed for the function call and the **`return`** linkage in the branches.

Exercise 5.33

Consider the following definition of a factorial function, which is slightly different from the one given above:

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

Compile this function and compare the resulting code with that produced for `factorial`. Explain any differences you find. Does either program execute more efficiently than the other?

Exercise 5.34

Compile the iterative factorial function

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

Annotate the resulting code, showing the essential difference between the code for iterative and recursive versions of `factorial` that makes one process build up stack space and the other run in constant stack space.

Exercise 5.35

What order of evaluation does our compiler produce for operands of a combination? Is it left-to-right, right-to-left, or some other order? Where in the compiler is this order determined? Modify the compiler so that it produces some other order of evaluation. (See the discussion of order of evaluation for the explicit-control evaluator in section 5.4.1.) How does changing the order of operand evaluation affect the efficiency of the code that constructs the argument list?

Exercise 5.36

One way to understand the compiler's `preserving` mechanism for optimizing stack usage is to see what extra operations would be generated if we did not use this idea. Modify `preserving` so that it always generates the `save` and `restore` operations. Compile some simple expressions and identify the unnecessary stack operations that are generated. Compare the code to that generated with the `preserving` mechanism intact.

Exercise 5.37

Our compiler is clever about avoiding unnecessary stack operations, but it is not clever at all when it comes to compiling calls to the primitive functions of the language in terms of the primitive operations supplied by the machine. For example, consider how much code is compiled to compute `a + 1`: The code sets up an argument list in `argl`, puts the primitive

addition function (which it finds by looking up the symbol + in the environment) into `fun`, and tests whether the function is primitive or compound. The compiler always generates code to perform the test, as well as code for primitive and compound branches (only one of which will be executed). We have not shown the part of the controller that implements primitives, but we presume that these instructions make use of primitive arithmetic operations in the machine's data paths. Consider how much less code would be generated if the compiler could *open-code* primitives—that is, if it could generate code to directly use these primitive machine operations. The expression `a + 1` might be compiled into something as simple as[42]

```
assign("val", op("lookup-variable-value"), constant("a"), reg("env")),
assign("val", op("+"), reg("val"), constant(1)),
```

In this exercise we will extend our compiler to support open coding of selected primitives. Special-purpose code will be generated for calls to these primitive functions instead of the general function-application code. In order to support this, we will augment our machine with special argument registers `arg1` and `arg2`. The primitive arithmetic operations of the machine will take their inputs from `arg1` and `arg2`. The results may be put into `val`, `arg1`, or `arg2`. The compiler must be able to recognize the application of an open-coded primitive in the source program. We will augment the dispatch in the `compile` function to recognize the names of these primitives in addition to the reserved words (the special forms) it currently recognizes.[43] For each special form our compiler has a code generator. In this exercise we will construct a family of code generators for the open-coded primitives.

a. The open-coded primitives, unlike the special forms, all need their operands evaluated. Write a code generator `spread_arguments` for use by all the open-coding code generators. `Spread_arguments` should take an operand list and compile the given operands targeted to successive argument registers. Note that an operand may contain a call to an open-coded primitive, so argument registers will have to be preserved during operand evaluation.

b. For each of the primitive functions `=`, `*`, `-`, and `+`, write a code generator that takes a combination with that operator, together with a target and a linkage descriptor, and produces code to spread the arguments into the registers and then perform the operation targeted to the given target with the given linkage. You need only handle expressions with two operands. Make `compile` dispatch to these code generators.

c. Try your new compiler on the `factorial` example. Compare the resulting code with the result produced without open coding.

[42] We have used the same symbol + here to denote both the source-language function and the machine operation. In general there will not be a one-to-one correspondence between primitives of the source language and primitives of the machine.

[43] Making the primitives into reserved words is in general a bad idea, since a user cannot then rebind these names to different functions. Moreover, if we add reserved words to a compiler that is in use, existing programs that define functions with these names will stop working. See exercise 5.43 for ideas on how to avoid this problem.

d. Extend your code generators for + and * so that they can handle expressions with arbitrary numbers of operands. An expression with more than two operands will have to be compiled into a sequence of operations, each with only two inputs.

5.5.6 Lexical Addressing

One of the most common optimizations performed by compilers is the optimization of variable lookup. Our compiler, as we have implemented it so far, generates code that uses the `lookup_variable_value` operation of the evaluator machine. This searches for a variable by comparing it with each variable that is currently bound, working frame by frame outward through the run-time environment. This search can be expensive if the frames are deeply nested or if there are many variables. For example, consider the problem of looking up the value of `x` while evaluating the expression `x * y * z` in an application of the function that is returned by

```
let x = 3;
let y = 4;
(a, b, c, d, e) => {
    let y = a * b * x;
    let z = c + d + x;
    return x * y * z;
}
```

Since a **`let`** expression is just syntactic sugar for a **`function`** `declaration expression` combination, this expression is equivalent to

```
((x, y) =>
    ((a, b, c, d, e) =>
        ((y, z) => x * y * z)(a * b * x, c + d + x)))(3, 4)
```

Each time `lookup_variable_value` searches for `x`, it must determine that the symbol `x` is not `eq?` to `y` or `z` (in the first frame), nor to `a`, `b`, `c`, `d`, or `e` (in the second frame). We will assume, for the moment, that our programs do not use `define`—that variables are bound only with `lambda`. Because our language is lexically scoped, the run-time environment for any expression will have a structure that parallels the lexical structure of the program in which the expression appears.[44] Thus, the compiler can know, when it analyzes the above expression, that each time the function is applied the variable `x` in `x * y * z` will be found two frames out from the current frame and will be the first variable in that frame.

We can exploit this fact by inventing a new kind of variable-lookup operation, `lexical_address_lookup`, that takes as arguments an environment and a *lexical address* that consists of two numbers: a *frame number*, which specifies how many frames to pass over, and a *displacement number*, which

[44] This is not true if we allow internal definitions, unless we scan them out. See exercise 5.42.

specifies how many variables to pass over in that frame. `Lexical_address_lookup` will produce the value of the variable stored at that lexical address relative to the current environment. If we add the `lexical_address_lookup` operation to our machine, we can make the compiler generate code that references variables using this operation, rather than `lookup_variable_value`. Similarly, our compiled code can use a new `lexical_address_set` operation instead of `set_variable_value`.

In order to generate such code, the compiler must be able to determine the lexical address of a variable it is about to compile a reference to. The lexical address of a variable in a program depends on where one is in the code. For example, in the following program, the address of `x` in expression e1 is (2,0)—two frames back and the first variable in the frame. At that point `y` is at address (0,0) and `c` is at address (1,2). In expression e2, `x` is at (1,0), `y` is at (1,1), and `c` is at (0,2).

```
((x, y) =>
    ((a, b, c, d, e) =>
        ((y, z) => e1)(e2, c + d + x)))(3, 4)
```

One way for the compiler to produce code that uses lexical addressing is to maintain a data structure called a *compile-time environment*. This keeps track of which variables will be at which positions in which frames in the run-time environment when a particular variable-access operation is executed. The compile-time environment is a list of frames, each containing a list of variables. (There will of course be no values bound to the variables, since values are not computed at compile time.) The compile-time environment becomes an additional argument to `compile` and is passed along to each code generator. The top-level call to `compile` uses an empty compile-time environment. When a `lambda` body is compiled, `compile_lambda_body` extends the compile-time environment by a frame containing the function's parameters, so that the sequence making up the body is compiled with that extended environment. At each point in the compilation, `compile_variable` and `compile_assignment` use the compile-time environment in order to generate the appropriate lexical addresses.

Exercises 5.38 through 5.42 describe how to complete this sketch of the lexical-addressing strategy in order to incorporate lexical lookup into the compiler. Exercise 5.43 describes another use for the compile-time environment.

Exercise 5.38

Write a function `lexical_address_lookup` that implements the new lookup operation. It should take two arguments—a lexical address and a run-time environment—and return the value of the variable stored at the specified lexical address. `Lexical_address_lookup` should signal an error if the value of the variable is the symbol `*unassigned*`.[45] Also write a function

[45]This is the modification to variable lookup required if we implement the scanning method to eliminate internal definitions (exercise 5.42). We will need to eliminate these definitions in order for lexical addressing to

`lexical_address_set` that implements the operation that changes the value of the variable at a specified lexical address.

Exercise 5.39

Modify the compiler to maintain the compile-time environment as described above. That is, add a compile-time-environment argument to `compile` and the various code generators, and extend it in `compile_lambda_body`.

Exercise 5.40

Write a function `find_variable` that takes as arguments a variable and a compile-time environment and returns the lexical address of the variable with respect to that environment. For example, in the program fragment that is shown above, the compile-time environment during the compilation of expression e_1 is `list(list("y", "z"), list("a", "b", "c", "d", "e"), list("x", "y"))`. `Find_variable` should produce

```
find_variable("c", list(list("y", "z"), list("a", "b", "c", "d", "e"), list("x", "y"
      [1, 2]

find_variable("x", list(list("y", "z"), list("a", "b", "c", "d", "e"), list("x", "y"
      [2, 0]

find_variable("w", list(list("y", "z"), list("a", "b", "c", "d", "e"), list("x", "y"
      not-found
```

Exercise 5.41

Using `find_variable` from exercise 5.40, rewrite `compile_variable` and `compile_assignment` to output lexical-address instructions. In cases where `find-variable` returns `not_found` (that is, where the variable is not in the compile-time environment), you should have the code generators use the evaluator operations, as before, to search for the binding. (The only place a variable that is not found at compile time can be is in the global environment, which is part of the run-time environment but is not part of the compile-time environment.[46] Thus, if you wish, you may have the evaluator operations look directly in the global environment, which can be obtained with the operation `op("get_global_environment")`, instead of having them search the whole run-time environment found in `env`.) Test the modified compiler on a few simple cases, such as the nested `lambda` combination at the beginning of this section.

work.

[46] Lexical addresses cannot be used to access variables in the global environment, because these names can be defined and redefined interactively at any time. With internal definitions scanned out, as in exercise 5.42, the only definitions the compiler sees are those at top level, which act on the global environment. Compilation of a definition does not cause the defined name to be entered in the compile-time environment.

Exercise 5.42

We argued in section 4.1.6 that internal definitions for block structure should not be considered 'real' `define`s. Rather, a function body should be interpreted as if the internal variables being defined were installed as ordinary `lambda` variables initialized to their correct values using assignment. Section 4.1.6 and exercise 4.8 showed how to modify the metacircular interpreter to accomplish this by scanning out internal definitions. Modify the compiler to perform the same transformation before it compiles a function body.

Exercise 5.43

In this section we have focused on the use of the compile-time environment to produce lexical addresses. But there are other uses for compile-time environments. For instance, in exercise d. we increased the efficiency of compiled code by open-coding primitive functions. Our implementation treated the names of open-coded functions as reserved words. If a program were to rebind such a name, the mechanism described in exercise d. would still open-code it as a primitive, ignoring the new binding. For example, consider the function

```
(+ * a b x y) => (a * x) + (b * y)
```

which computes a linear combination of `x` and `y`. We might call it with arguments `+matrix`, `*matrix`, and four matrices, but the open-coding compiler would still open-code the `+` and the `*` in `(a * x) + (b * y)` as primitive `+` and `*`. Modify the open-coding compiler to consult the compile-time environment in order to compile the correct code for expressions involving the names of primitive functions. (The code will work correctly as long as the program does not `define` or assignment these names.)

5.5.7 Interfacing Compiled Code to the Evaluator

We have not yet explained how to load compiled code into the evaluator machine or how to run it. We will assume that the explicit-control-evaluator machine has been defined as in section 5.4.4, with the additional operations specified in footnote 37. We will implement a function `compile-and-go` that compiles a Scheme expression, loads the resulting object code into the evaluator machine, and causes the machine to run the code in the evaluator global environment, print the result, and enter the evaluator's driver loop. We will also modify the evaluator so that interpreted expressions can call compiled functions as well as interpreted ones. We can then put a compiled function into the machine and use the evaluator to call it:

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

;;; EC-Eval value:
ok

;;; EC-Eval input:
(factorial 5)
;;; EC-Eval value:
120
```

To allow the evaluator to handle compiled functions (for example, to evaluate the call to `factorial` above), we need to change the code at `apply_dispatch` (section 5.4.1) so that it recognizes compiled functions (as distinct from compound or primitive functions) and transfers control directly to the entry point of the compiled code:[47]

```
"apply_dispatch",
    test(op("primitive_procedure"), reg("proc")),
    branch(label("primitive_apply")),
    test(op("compound_procedure"), reg("proc")),
    branch(label("compound_apply")),
    test(op("compiled_procedure"), reg("proc")),
    branch(label("compiled_apply")),
    go_to(label("unknown_procedure_type")),

"compiled_apply",
    restore("continue"),
    assign("val", list(op("compiled_procedure_entry"), reg("proc"))),
    go_to(reg("val")),
```

Note the restore of **`continue`** at `compiled_apply`. Recall that the evaluator was arranged so that at `apply_dispatch`, the continuation would be at the top of the stack. The compiled code entry point, on the other hand, expects the continuation to be in **`continue`**, so **`continue`** must be restored before the compiled code is executed.

To enable us to run some compiled code when we start the evaluator machine, we add a `branch` instruction at the beginning of the evaluator machine, which causes the machine to go to a new entry point if the `flag` register is set.[48]

[47] Of course, compiled functions as well as interpreted functions are compound (nonprimitive). For compatibility with the terminology used in the explicit-control evaluator, in this section we will use 'compound' to mean interpreted (as opposed to compiled).

[48] Now that the evaluator machine starts with a `branch`, we must always initialize the `flag` register before

```
branch(label("external_entry")), // branches if flag is set
"read_eval_print_loop",
perform(op("initialize-stack")),
⟨...⟩
```

`External_entry` assumes that the machine is started with `val` containing the location of an instruction sequence that puts a result into `val` and ends with `go_to(reg("continue"))`. Starting at this entry point jumps to the location designated by `val`, but first assigns **`continue`** so that execution will return to `print_result`, which prints the value in `val` and then goes to the beginning of the evaluator's read-eval-print loop.[49]

```
"external_entry",
    perform(op("initialize_stack")),
    assign("env", op("get_global_environment")),
    assign("continue", label("print_result")),
    go_to(reg("val")),
```

Now we can use the following function to compile a function definition, execute the compiled code, and run the read-eval-print loop so we can try the function. Because we want the compiled code to return to the location in **`continue`** with its result in `val`, we compile the expression with a target of `val` and a linkage of **`return`**. In order to transform the object code produced by the compiler into executable instructions for the evaluator register machine, we use the function `assemble` from the register-machine simulator (section 5.2.2). We then initialize the `val` register to point to the list of instructions, set the `flag` so that the evaluator will go to `external_entry`, and start the evaluator.

---

starting the evaluator machine. To start the machine at its ordinary read-eval-print loop, we could use

```
function start_eceval() {
    the_global_environment = setup_environment();
    set_register_contents(eceval, "flag", false);
    return start(eceval);
}
```

[49] Since a compiled function is an object that the system may try to print, we also modify the system print operation `user_print` (from section 4.1.4) so that it will not attempt to print the components of a compiled function:

```
function user_print(object) {
    if (compound_procedure(object)) {
        display(list(
            "compound_procedure",
            procedure_parameters(object),
            procedure_body(object),
            "<compiler-env>"));
    } else if (compiled_procedure(object)) {
        display("<compiler-procedure>");
    } else {
        display(object);
    }
}
```

```
function compile_and_go(expression) {
    const instructions = assemble(statements(compile(expression, "val", "return")),
    the-global-environment = setup-environment();
    set_register_contents("eceval", "val", instructions);
    set_register_contents("eceval", "flag", true);
    return start("eceval");
}
```

If we have set up stack monitoring, as at the end of section 5.4.4, we can examine the stack usage of compiled code:

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

(total-pushes = 0 maximum-depth = 0)
;;; EC-Eval value:
ok

;;; EC-Eval input:
(factorial 5)
(total-pushes = 31 maximum-depth = 14)
;;; EC-Eval value:
120
```

Compare this example with the evaluation of `factorial(5)` using the interpreted version of the same function, shown at the end of section 5.4.4. The interpreted version required 144 pushes and a maximum stack depth of 28. This illustrates the optimization that results from our compilation strategy.

Interpretation and compilation

With the programs in this section, we can now experiment with the alternative execution strategies of interpretation and compilation.[50] An interpreter raises the machine to the level of the user program; a compiler lowers the user program to the level of the machine language. We can regard the Scheme language (or any programming language) as a coherent family of abstractions erected on the machine language. Interpreters are good for interactive program development and debugging because the steps of program execution are organized in terms of these abstractions, and are therefore more intelligible to the programmer. Compiled code can

[50] We can do even better by extending the compiler to allow compiled code to call interpreted functions. See exercise 5.46.

execute faster, because the steps of program execution are organized in terms of the machine language, and the compiler is free to make optimizations that cut across the higher-level abstractions.[51]

Compilers for popular languages, such as C and C++, put hardly any error-checking operations into running code, so as to make things run as fast as possible. As a result, it falls to programmers to explicitly provide error checking. Unfortunately, people often neglect to do this, even in critical applications where speed is not a constraint. Their programs lead fast and dangerous lives. For example, the notorious 'Worm' that paralyzed the Internet in 1988 exploited the UNIX™ operating system's failure to check whether the input buffer has overflowed in the finger daemon. (See Spafford 1989.)

The alternatives of interpretation and compilation also lead to different strategies for porting languages to new computers. Suppose that we wish to implement JavaScript for a new machine. One strategy is to begin with the explicit-control evaluator of section 5.4 and translate its instructions to instructions for the new machine. A different strategy is to begin with the compiler and change the code generators so that they generate code for the new machine. The second strategy allows us to run any JavaScript program on the new machine by first compiling it with the compiler running on our original JavaScript system, and linking it with a compiled version of the run-time library.[52] Better yet, we can compile the compiler itself, and run this on the new machine to compile other JavaScript programs.[53] Or we can compile one of the interpreters of section 4.1 to produce an interpreter that runs on the new machine.

Exercise 5.44

By comparing the stack operations used by compiled code to the stack operations used by the evaluator for the same computation, we can determine the extent to which the compiler optimizes use of the stack, both in speed (reducing the total number of stack operations) and

[51] Independent of the strategy of execution, we incur significant overhead if we insist that errors encountered in execution of a user program be detected and signaled, rather than being allowed to kill the system or produce wrong answers. For example, an out-of-bounds array reference can be detected by checking the validity of the reference before performing it. The overhead of checking, however, can be many times the cost of the array reference itself, and a programmer should weigh speed against safety in determining whether such a check is desirable. A good compiler should be able to produce code with such checks, should avoid redundant checks, and should allow programmers to control the extent and type of error checking in the compiled code.

[52] Of course, with either the interpretation or the compilation strategy we must also implement for the new machine storage allocation, input and output, and all the various operations that we took as 'primitive' in our discussion of the evaluator and compiler. One strategy for minimizing work here is to write as many of these operations as possible in JavaScript and then compile them for the new machine. Ultimately, everything reduces to a small kernel (such as garbage collection and the mechanism for applying actual machine primitives) that is hand-coded for the new machine.

[53] This strategy leads to amusing tests of correctness of the compiler, such as checking whether the compilation of a program on the new machine, using the compiled compiler, is identical with the compilation of the program on the original JavaScript system. Tracking down the source of differences is fun but often frustrating, because the results are extremely sensitive to minuscule details.

in space (reducing the maximum stack depth). Comparing this optimized stack use to the performance of a special-purpose machine for the same computation gives some indication of the quality of the compiler.

a. Exercise 5.27 asked you to determine, as a function of n, the number of pushes and the maximum stack depth needed by the evaluator to compute $n!$ using the recursive factorial function given above. Exercise 5.14 asked you to do the same measurements for the special-purpose factorial machine shown in Figure 5.11. Now perform the same analysis using the compiled `factorial` function. Take the ratio of the number of pushes in the compiled version to the number of pushes in the interpreted version, and do the same for the maximum stack depth. Since the number of operations and the stack depth used to compute $n!$ are linear in n, these ratios should approach constants as n becomes large. What are these constants? Similarly, find the ratios of the stack usage in the special-purpose machine to the usage in the interpreted version. Compare the ratios for special-purpose versus interpreted code to the ratios for compiled versus interpreted code. You should find that the special-purpose machine does much better than the compiled code, since the hand-tailored controller code should be much better than what is produced by our rudimentary general-purpose compiler.

b. Can you suggest improvements to the compiler that would help it generate code that would come closer in performance to the hand-tailored version?

Exercise 5.45

Carry out an analysis like the one in exercise 5.44 to determine the effectiveness of compiling the tree-recursive Fibonacci function

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

compared to the effectiveness of using the special-purpose Fibonacci machine of figure 5.12. (For measurement of the interpreted performance, see exercise 5.28.) For Fibonacci, the time resource used is not linear in n; hence the ratios of stack operations will not approach a limiting value that is independent of n.

Exercise 5.46

This section described how to modify the explicit-control evaluator so that interpreted code can call compiled functions. Show how to modify the compiler so that compiled functions

can call not only primitive functions and compiled functions, but interpreted functions as well. This requires modifying `compile_procedure_call` to handle the case of compound (interpreted) functions. Be sure to handle all the same `target` and `linkage` combinations as in `compile_proc_appl`. To do the actual function application, the code needs to jump to the evaluator's `compound_apply` entry point. This label cannot be directly referenced in object code (since the assembler requires that all labels referenced by the code it is assembling be defined there), so we will add a register called `compapp` to the evaluator machine to hold this entry point, and add an instruction to initialize it:

```
assign("compapp", label("compound_apply")),
branch(label("external_entry")),
"read_eval_print_loop"
...
```

To test your code, start by defining a function `f` that calls a function `g`. Use `compile_and_go` to compile the definition of `f` and start the evaluator. Now, typing at the evaluator, define `g` and try to call `f`.

Exercise 5.47

The `compile_and_go` interface implemented in this section is awkward, since the compiler can be called only once (when the evaluator machine is started). Augment the compiler-interpreter interface by providing a `compile_and_run` primitive that can be called from within the explicit-control evaluator as follows:

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

;;; EC-Eval value:
ok

;;; EC-Eval input:
(factorial 5)
;;; EC-Eval value:
120
```

Exercise 5.48

As an alternative to using the explicit-control evaluator's read-eval-print loop, design a register

machine that performs a read-compile-execute-print loop. That is, the machine should run a loop that reads an expression, compiles it, assembles and executes the resulting code, and prints the result. This is easy to run in our simulated setup, since we can arrange to call the functions `compile` and `assemble` as 'register-machine operations.'

Exercise 5.49

Use the compiler to compile the metacircular evaluator of section 4.1 and run this program using the register-machine simulator. (To compile more than one definition at a time, you can package the definitions in a sequence.) The resulting interpreter will run very slowly because of the multiple levels of interpretation, but getting all the details to work is an instructive exercise.

Exercise 5.50

Develop a rudimentary implementation of JavaScript in C (or some other low-level language of your choice) by translating the explicit-control evaluator of section 5.4 into C. In order to run this code you will need to also provide appropriate storage-allocation routines and other run-time support.

Exercise 5.51

As a counterpoint to exercise 5.50, modify the compiler so that it compiles JavaScript functions into sequences of C instructions. Compile the metacircular evaluator of section 4.1 to produce a JavaScript interpreter written in C.

List of exercises

Solution To Exercises

Answer of exercise 2.1

```
function last_pair(items) {
    return is_null(tail(items))
           ? items
           : last_pair(tail(items));
}
```

Answer of exercise 2.2

```
// naive reverse (what is the runtime?)
function reverse(items) {
    return is_null(items)
           ? null
           : append(reverse(tail(items)),
                    pair(head(items), null));
}
```

```
// a better version
function reverse(items) {
    function reverse_iter(items, result) {
        return is_null(items)
               ? result
               : reverse_iter(tail(items),
                              pair(head(items), result));
    }
    return reverse_iter(items, null);
}
```

Answer of exercise 2.3

```
function first_denomination(coin_values) {
    return head(coin_values);
}
function except_first_denomination(coin_values) {
    return tail(coin_values);
}
function no_more(coin_values) {
    return is_null(coin_values);
}
```

The order of the list coin_values does not affect the answer given by any correct solution of the problem, because the given list represents an unordered collection of denominations.

Answer of exercise 2.4

a.

```
function brooks(f, items) {
    return is_null(items)
           ? f
           : brooks(f(head(items)), tail(items));
}
```

b.

```
function brooks_curried(items) {
    return brooks(head(items), tail(items));
}
```

c. The statement

```
brooks_curried(list(brooks_curried,
                    list(plus_curried, 3, 4)));
```

of course evaluates to 7, as does

d.

```
brooks_curried(list(brooks_curried,
                    list(brooks_curried,
                         list(plus_curried, 3, 4))));
```

Answer of exercise 2.5

```
function square_list(items) {
    return is_null(items)
           ? null
           : pair(square(head(items)),
                  square_list(tail(items)));
}
```

```
function square_list(items) {
    return map(square, items);
}
```

Answer of exercise 2.6

a. The result list is reversed in the first program because the argument list is traversed in the given order, from first to last, but squares are added successively to the front of the answer list via `pair`. The last element of the list is the last one to be added to the answer and thus ends up as the first element of the result list.

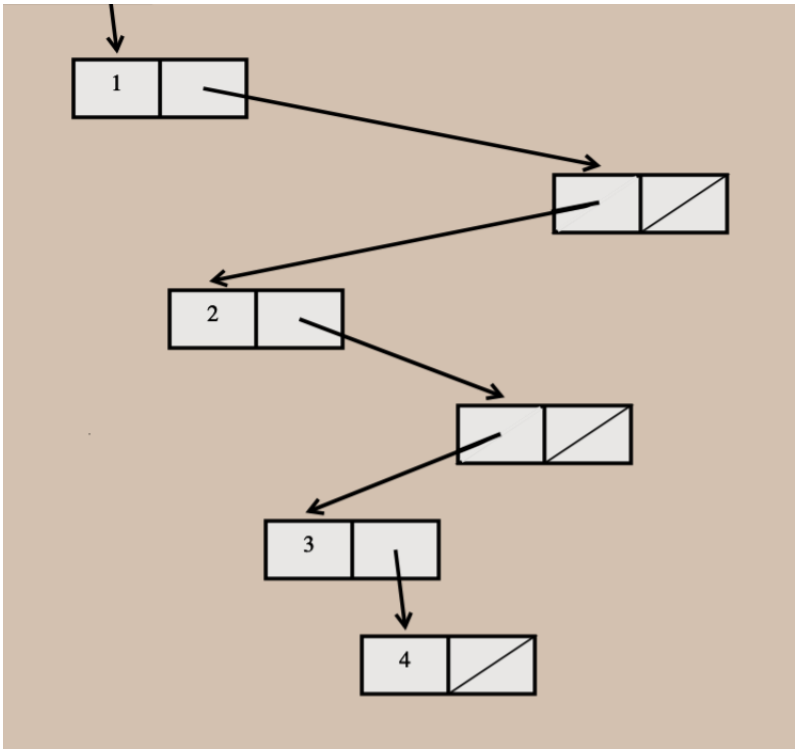
b. The second program makes things worse! The result is not even a list any longer, because the elements occupy the tail position of the result list and not the head position.

Answer of exercise 2.7

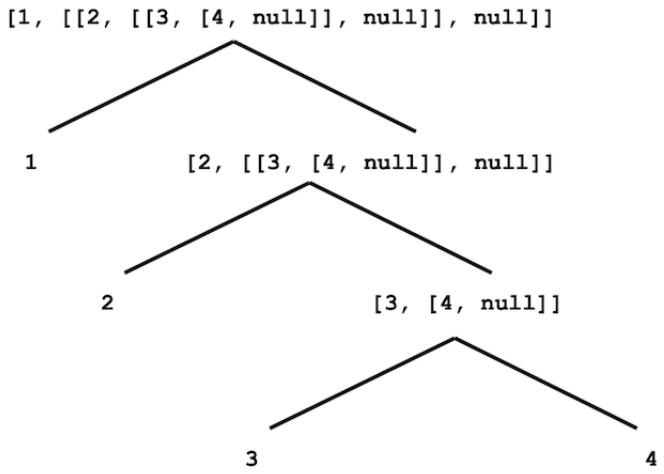
```
function for_each(fun, items) {
    if (is_null(items)){
        return undefined;
    } else {
        fun(head(items));
        for_each(fun, tail(items));
    }
}
```

Answer of exercise 2.8

a. `[1, [[2, [[3, [4, null]], null]], null]]`



b.



c.

Answer of exercise 2.9

```
- head(tail(head(tail(tail(the_first_list)))));

- head(head(the_second_list));

- head(tail(head(tail(head(tail(head(tail(head(
    tail(head(tail(the_third_list))))))))))));
```

Answer of exercise 2.10

a.

```
[1, [2, [3, [4, [5, [6, null]]]]]]
```

b.

```
[[1, [2, [3, null]]], [4, [5, [6, null]]]]
```

c.

```
[[1, [2, [3, null]]], [[4, [5, [6, null]]], null]]
```

Answer of exercise 2.11

```
function deep_reverse(items){
    return is_null(items)
           ? null
           : is_pair(items)
             ? append(deep_reverse(tail(items)),
                      pair(deep_reverse(head(items)),
                           null))
             : items;
}
```

Answer of exercise 2.12

```
function fringe(x) {
    return is_null(x)
           ? null
           : is_pair(x)
             ? append(fringe(head(x)), fringe(tail(x)))
             : list(x);
}
```

Answer of exercise 2.13

a.

```
function left_branch(m) {
    return head(m);
}
function right_branch(m) {
    return head(tail(m));
}
function branch_length(b) {
    return head(b);
}
function branch_structure(b) {
    return head(tail(b));
}
```

b.

```
function is_weight(x){
    return is_number(x);
}
function total_weight(x) {
    return is_weight(x)
        ? x
        : total_weight(branch_structure(
                       left_branch(x))) +
          total_weight(branch_structure(
```

```
                           right_branch(x)));
}
```

c.

```
function is_balanced(x) {
    return is_weight(x) ||
        ( is_balanced(branch_structure(
                                    left_branch(x))) &&
          is_balanced(branch_structure(
                                    right_branch(x))) &&
         total_weight(branch_structure(
                                left_branch(x)))
         * branch_length(left_branch(x))
         ===
         total_weight(branch_structure(
                                right_branch(x)))
         * branch_length(right_branch(x))
        );
}
```

d. With this alternative representation, the accessor functions for mobile and branch need to change as follows:

```
function left_branch(m) {
  return head(m);
}
function right_branch(m) {
  return tail(m);
}
function branch_length(b) {
  return head(b);
}
function branch_structure(b) {
  return tail(b);
}
```

Answer of exercise 2.14

Directly:

```
function square_tree(tree) {
    return is_null(tree)
           ? null
           : ! is_pair(tree)
             ? square(tree)
             : pair(square_tree(head(tree)),
                    square_tree(tail(tree)));
}
```

The version using map:

```
function square_tree(tree) {
    return map(sub_tree => ! is_pair(sub_tree)
                           ? square(sub_tree)
                           : square_tree(sub_tree),
               tree);
}
```

Answer of exercise 2.15

```
function tree_map(f, tree) {
    return map(sub_tree => is_null(sub_tree)
                           ? null
                           : is_pair(sub_tree)
                             ? tree_map(f, sub_tree)
                             : f(sub_tree),
               tree);
}
```

Answer of exercise 2.16

```
function subsets(s) {
    if (is_null(s)) {
        return list(null);
    } else {
        const rest = subsets(tail(s));
        return append(rest, map(x => pair(head(s), x), rest));
    }
}
```

The argument starts in a similar way as the argument for the function cc in section **??**: A subset either contains the first element e of the given set, or it doesn't. If it doesn't, the problem becomes strictly smaller: Compute all subsets of the tail of the list that represents the given set. If it does, it must result from adding e to a subset that doesn't contain e. In the end, we need to append both lists of subsets to obtain the list of all subsets.

Answer of exercise 2.17

```
function map(f, sequence) {
    return accumulate((x, y) => pair(p(x), y),
                      null,
                      sequence);
}

function append(seq1, seq2) {
    return accumulate(pair, seq2, seq1);
```

```
}

function length(sequence) {
    return accumulate((x, y) => y + 1,
                      0,
                      sequence);
}
```

Answer of exercise 2.18

```
function horner_eval(x, coefficient_sequence) {
    return accumulate((this_coeff, higher_terms) =>
                      x * higher_terms + this_coeff,
                      0,
                      coefficient_sequence);
}
```

Answer of exercise 2.19

```
function count_leaves(t) {
    return accumulate((leaves, total) => leaves + total,
                      0,
                      map(sub_tree => is_pair(sub_tree)
                             ? count_leaves(sub_tree)
                             : 1,
                          t));
}
```

Answer of exercise 2.20

```
function accumulate_n(op, init, seqs) {
    return is_null(head(seqs))
           ? null
           : pair(accumulate(op, init, map(x => head(x), seqs)),
                  accumulate_n(op, init, map(x => tail(x), seqs)));
}
```

Answer of exercise 2.21

```
function matrix_times_vector(m, v) {
    return map(row => dot_product(row, v), m);
}

function transpose(mat) {
    return accumulate_n(pair, null, mat);
}
function matrix_times_matrix(n, m) {
```

```
    const cols = transpose(m);
    return map(x => map(y => dot_product(x, y), cols), n);
}
```

Answer of exercise 2.22

We can guarantee that `fold_right` and `fold_left` produce the same values for any sequence, if we require that op is commutative and associative.

```
fold_right(plus, 0, list(1, 2, 3));
```

```
fold_left(plus, 0, list(1, 2, 3));
```

Answer of exercise 2.23

```
function reverse(sequence) {
    return fold_right((x, y) => append(y, list(x)),
                      null, sequence);
}
```

```
function reverse(sequence) {
    return fold_left((x, y) => pair(y, x), null, sequence);
}
```

Answer of exercise 2.24

```
function unique_pairs(n) {
    return flatmap(i => map(j => list(i, j),
                            enumerate_interval(1, i-1)),
                   enumerate_interval(1, n));
}
function prime_sum_pairs(n) {
    return map(make_pair_sum,
               filter(is_prime_sum,
                      unique_pairs(n)));
}
```

Answer of exercise 2.25

```
function unique_triples(n) {
    return flatmap(i => flatmap(j => map(k => list(i, j, k),
                                         enumerate_interval(1, j-1)),
                                enumerate_interval(1, i-1)),
                   enumerate_interval(1, n));
}
function plus(x, y) {
    return x + y;
```

```
}
function triples_that_sum_to(s, n) {
    return filter(items => accumulate(plus, 0, items) === s,
                  unique_triples(n));
}
```

Answer of exercise 2.26

```
function adjoin_position(row, col, rest) {
    return pair(pair(row, col), rest);
}
```

```
const empty_board = null;
```

```
function is_safe(k, positions) {
    const first_row = head(head(positions));
    const first_col = tail(head(positions));
    return accumulate((pos, so_far) => {
                          const row = head(pos);
                          const col = tail(pos);
                          return so_far &&
                                 first_row - first_col !==
                                 row - col &&
                                 first_row + first_col !==
                                 row + col &&
                                 first_row !== row;
                      },
                      true,
                      tail(positions));
}
```

Putting it all together:

```
// click here to see the solution
```

Answer of exercise 2.27

Louis's program re-evaluates the application `queen_cols(k - 1))` in each iteration of `flatmap`, which happens n times for each k. That means overall Louis's program will solve the puzzle in a time of about $n^n T$ if the program in exercise 2.26 solves the puzzle in time T.

Answer of exercise 2.28

```
function up_split(painter, n) {
    if (n === 0) {
        return painter;
    } else {
        const smaller = up_split(painter, n - 1);
        return stack(beside(smaller, smaller), painter);
    }
}
```

Answer of exercise 2.29

```
function split(identity_op, smaller_op) {
    function rec_split(painter, n) {
        if (n===0) {
            return painter;
        } else {
            const smaller = rec_split(painter, n - 1);
            return identity_op(painter,
                               smaller_op(smaller, smaller));
        }
    }
    return rec_split;
}
```

```
const right_split = split(beside, stack);

show(right_split(heart, 4));
```

Answer of exercise 2.30

```
function make_vect(x, y) {
    return pair(x, y);
}
function xcor_vect(vector) {
    return head(vector);
}
function ycor_vect(vector) {
    return tail(vector);
}
function scale_vect(factor, vector) {
    return make_vect(factor * xcor_vect(vector),
                     factor * ycor_vect(vector));
}
function add_vect(vector1, vector2) {
    return make_vect(xcor_vect(vector1)
                     + xcor_vect(vector2),
                     ycor_vect(vector1)
```

```
                    + ycor_vect(vector2));
}
function sub_vect(vector1, vector2) {
    return make_vect(xcor_vect(vector1)
                    - xcor_vect(vector2),
                    ycor_vect(vector1)
                    - ycor_vect(vector2));
}
```

Answer of exercise 2.31

a.

```
function make_frame(origin, edge1, edge2) {
    return list(origin, edge1, edge2);
}
function origin_frame(frame) {
    return list_ref(frame, 0);
}
function edge1_frame(frame) {
    return list_ref(frame, 1);
}
function edge2_frame(frame) {
    return list_ref(frame, 2);
}
```

b.

```
function make_frame(origin, edge1, edge2) {
    return pair(origin, pair(edge1, edge2));
}
function origin_frame(frame) {
    return head(frame);
}
function edge1_frame(frame) {
    return head(tail(frame));
}
function edge2_frame(frame) {
    return tail(tail(frame));
}
```

Answer of exercise 2.32

```
function make_segment(v_start, v_end) {
    return pair(v_start, v_end);
}
function start_segment(v) {
    return head(v);
}
```

```
function end_segment(v) {
    return tail(v);
}
```

Answer of exercise 2.33

a. The painter that draws the outline of the designated frame.

```
const outline_start_1 = make_vect(0.0, 0.0);
const outline_end_1 = make_vect(1.0, 0.0);
const outline_segment_1 = make_segment(outline_start_1,
                                       outline_end_1);
const outline_start_2 = make_vect(1.0, 0.0);
const outline_end_2 = make_vect(1.0, 1.0);
const outline_segment_2 = make_segment(outline_start_2,
                                       outline_end_2);
const outline_start_3 = make_vect(1.0, 1.0);
const outline_end_3 = make_vect(0.0, 1.0);
const outline_segment_3 = make_segment(outline_start_3,
                                       outline_end_3);
const outline_start_4 = make_vect(0.0, 1.0);
const outline_end_4 = make_vect(0.0, 0.0);
const outline_segment_4 = make_segment(outline_start_4,
                                       outline_end_4);
const outline_painter = segments_to_painter(
                            list(outline_segment_1,
                                 outline_segment_2,
                                 outline_segment_3,
                                 outline_segment_4));
```

b. The painter that draws an 'X' by connecting opposite corners of the frame.

```
const x_start_1 = make_vect(0.0, 0.0);
const x_end_1 = make_vect(1.0, 1.0);
const x_segment_1 = make_segment(x_start_1,
                                 x_end_1);
const x_start_2 = make_vect(1.0, 0.0);
const x_end_2 = make_vect(0.0, 1.0);
const x_segment_2 = make_segment(x_start_2,
                                 x_end_2);
const x_painter = segments_to_painter(
                      list(x_segment_1,
                           x_segment_2));
```

c. The painter that draws a diamond shape by connecting the midpoints of the sides of the frame.

```
    const diamond_start_1 = make_vect(0.5, 0.0);
    const diamond_end_1 = make_vect(1.0, 0.5);
    const diamond_segment_1 = make_segment(diamond_start_1,
                                           diamond_end_1);
    const diamond_start_2 = make_vect(1.0, 0.5);
    const diamond_end_2 = make_vect(0.5, 1.0);
const diamond_segment_2 = make_segment(diamond_start_2,
                                       diamond_end_2);
const diamond_start_3 = make_vect(0.5, 1.0);
const diamond_end_3 = make_vect(0.0, 0.5);
const diamond_segment_3 = make_segment(diamond_start_3,
                                       diamond_end_3);
const diamond_start_4 = make_vect(0.0, 0.5);
const diamond_end_4 = make_vect(0.5, 0.0);
const diamond_segment_4 = make_segment(diamond_start_4,
                                       diamond_end_4);
const diamond_painter = segments_to_painter(
                             list(diamond_segment_1,
                                  diamond_segment_2,
                                  diamond_segment_3,
                                  diamond_segment_4));
```

Answer of exercise 2.34

a. The transformation `flip_horiz` :

```
function flip_horiz(painter) {
    return transform_painter(painter,
                make_vect(1.0, 0.0),  // new origin
                make_vect(0.0, 0.0),  // new end of edge1
                make_vect(1.0, 1.0)); // new end of edge2
}
```

b. The transformation `rotate180` :

```
function rotate180(painter) {
    return transform_painter(
                painter,
                make_vect(1.0, 1.0),  // new origin
                make_vect(0.0, 1.0),  // new end of edge1
                make_vect(1.0, 0.0)); // new end of edge2
}
```

c. The transformation `rotate270` :

```
    function rotate270(painter) {
        return transform_painter(
                    painter,
                    make_vect(0.0, 1.0),  // new origin
                    make_vect(0.0, 0.0),  // new end of edge1
                    make_vect(1.0, 0.0)); // new end of edge2
}
```

Answer of exercise 2.35

a. First the direct method:

```
    function stack(painter1, painter2) {
        const split_point = make_vect(0.0, 0.5);
        const paint_upper =
            transform_painter(painter1,
                              split_point,
                              make_vect(1.0, 0.5),
                          make_vect(0.0, 1.0));
    const paint_lower =
        transform_painter(painter2,
                          make_vect(0.0, 0.0),
                          make_vect(1.0, 0.0),
                          split_point);
    return frame => {
                paint_upper(frame);
                paint_lower(frame);
           };
}
```

b. Now the version with rotation and beside:

```
    function stack(painter1, painter2) {
        return rotate270(beside(rotate90(painter1),
                                rotate90(painter2)));
    }
```

Answer of exercise 2.36

```
// Click here to play with any abstraction
// used for square_limit
```

Answer of exercise 2.37

a. *Explain what was done above. Why can't we assimilate the predicates* `is_number` *and* `is_same_variable` *into the data-directed dispatch?*

The operator symbols come very handy as 'type' keys in the operator table. For numbers and variables, there aren't such obvious keys, although we could introduce names for those types of expressions, as well, if we change the way expressions are represented as lists.

b. *Write the functions for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.*

```
function deriv_sum(operands, variable) {
    return make_sum(deriv(addend(operands), variable),
                    deriv(augend(operands), variable));
}
function deriv_product(operands, variable) {
    return make_sum(make_product(multiplier(operands),
                                 deriv(multiplicand(operands),
                                       variable)),
                    make_product(deriv(multiplier(
                                                operands),
                                       variable),
                                 multiplicand(operands)));
}
function install_deriv() {
    put("deriv", "+", deriv_sum);
    put("deriv", "*", deriv_product);
    return "done";
}
install_deriv();
```

c. *Choose any additional differentiation rule that you like, such as the one for exponents (Exercise* ??*), and install it in this data-directed system.*

```
function deriv_exponentiation(operands, variable) {
    const bas = base(operands);
    const exp = exponent(operands);
    return make_product(exp,
                make_product(make_exponentiation(bas, make_sum(exp, -1)),
                             deriv(bas, variable)));
}
function install_exponentiation_extension() {
    put("deriv", "**", deriv_exponentiation);
    return "done";
}
install_exponentiation_extension();
```

d. *In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the functions in the opposite way, so that the dispatch line in* `deriv` *looked like*

```
get(operator(exp), "deriv")(operands(exp), variable);
```

What corresponding changes to the derivative system are required?

We would need to change the order of arguments in the installation procedure for the differentiation library:

```
put("+", "deriv", deriv_sum);
put("*", "deriv", deriv_product);
put("**", "deriv", deriv_exponentiation);
```

Answer of exercise 2.38

a. *Implement for headquarters a* `get_record` *function that retrieves a specified employee's record from a specified personnel file. The function should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?*

We are tagging each division's file with a unique identifier for the division, using the tagging functions in section 2.2.2. We assume that each division provides an implementation of the `get_record` function and installs it in the company-wide operations table.

```
function make_insatiable_file(division, file) {
    return pair(division, file);
}
function insatiable_file_division(insatiable_file) {
    return head(insatiable_file);
}
function insatiable_file_content(insatiable_file) {
    return tail(insatiable_file);
}
function get_record(employee_name, insatiable_file) {
    const the_division
        = insatiable_file_division(insatiable_file);
    const division_record = get("get_record", the_division)
                            (employee_name,
                             insatiable_file_content(
                                 insatiable_file);
    return record !== undefined
           ? attach_tag(the_division, division_record)
           : undefined;
}
```

b. *Implement for headquarters a* `get_salary` *function that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?*

Every division needs to implement functions such as `get_salary` and install them in Insatiable's operations table. Then, Insatiable's function `get_salary` can look like this:

```
function make_insatiable_record(division, record) {
    return pair(division, record);
}
function insatiable_record_division(insatiable_record) {
    return head(insatiable_record);
}
function insatiable_record_content(insatiable_record) {
    return tail(insatiable_record);
}
function get_salary(insatiable_record) {
    const the_division =
        insatiable_record_division(insatiable_record);
    return get("get_salary", the_division)
           (insatiable_record_content);
}
```

Note that we rely on the fact that any employee record that gets returned by `get_record` is tagged with its division, which is used in the generic function `get_salary` to retrieve the correct implementation from the operation table.

c. *Implement for headquarters a* `find_employee_record` *function. This should search all the divisions' files for the record of a given employee and return the record. Assume that this function takes as arguments an employee's name and a list of all the divisions' files.*

```
function find_employee_record(employee_name,
                              personnel_files) {
    if (is_null(personell_files)) {
        return undefined;
    } else {
        const insatiable_record
            = get_record(employee_name,
                         head(personell_files));
        return insatiable_record !== undefined
               ? insatiable_record
               : find_employee_record(employee_name,
                              tail(personell_files));
    }
}
```

d. *When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?*

We would need to do the following, for each newly acquired company:

- Decide a name to be used as tag for any data item related to the new division.
- Write all division-specific functions such as `get_salary` and install them in the company-wide operations table using the division tag.
- Add the employee files to the list of `personell_files`. Note that this is a 'destructive' operation—similar to the extension of operations tables—in that the data structure is permanently and irrevocably modified; section 3.3 explains this concept in detail.

Answer of exercise 2.39

```
function make_from_mag_ang(r, a) {
    function dispatch(op) {
        return op === "real_part"
            ? r * math_cos(a)
            : op === "imag_part"
              ? r * math_sin(a)
              : op === "magnitude"
                ? r
                : op === "angle"
                  ? a
                  : Error("Unknown op in make_from_real_imag",
                          op);
    }
    return dispatch;
}
```

Answer of exercise 2.40

- *Generic operations with explicit dispatch*: For every new type, we need to touch every generic interface function, and add a new case.
- *Data-directed style*: Here the implementation of the generic interface functions can be neatly packaged in 'install' libraries for each new type. We can also have 'install' libraries for new operations.
- *Message-passing-style*: Like in the data-directed style, we need to write a library for each new type. In this case, the library consists of a dispatch function with a case for every generic interface function.

Overall, it's probably best to use a data-directed style when we need to frequently add new operations, and message-passing, when we frequently add new types.

Answer of exercise 3.1

```
function make_accumulator(current) {
    function add(arg) {
        current = current + arg;
        return current;
    }
    return add;
}
```

Answer of exercise 3.2

```
const s = make_monitored(math_sqrt);
s(100);
display(s("how many calls"));
s(5);
display(s("how many calls"));
```

```
function make_monitored(f) {
    let counter = 0; //initialized to 0
    function mf(cmd) {
        if (cmd === "how many calls") {
            return counter;
        } else if (cmd === "reset count") {
            counter = 0;
            return counter;
        } else {
            counter = counter + 1;
            return f(cmd);
        }
    }
    return mf;
}
```

Answer of exercise 3.3

```
function make_account(balance, p) {
    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }
    function deposit(amount) {
        balance = balance + amount;
```

```
        return balance;
    }
    function dispatch(m, q) {
        if (p === q) {
            if (m === "withdraw") {
                return withdraw;
            } else if (m === "deposit") {
                return deposit;
            } else {
                return "Unknown request - make_account";
            }
        } else {
            return q => "Incorrect Password";
        }
    }
    return dispatch;
}

const a = make_account(100, "eva");
(a("withdraw", "eva"))(50); //withdraws 50
(a("withdraw", "ben"))(40); //incorrect password
```

Answer of exercise 3.4

```
function make_account(balance, p) {

    let invalid_attempts = 0; //initializes to 0

    function withdraw(amount) {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    }

    function deposit(amount) {
        balance = balance + amount;
        return balance;
    }

    function call_the_cops() {
        return "calling the cops as you have exceeded " +
               "the max no of failed attempts";
    }

    function dispatch(m, q) {
```

```
        if (invalid_attempts <= 7) {
            if (p === q) {
                if (m === "withdraw") {
                    return withdraw;
                } else if (m === "deposit") {
                    return deposit;
                } else {
                    return "Unknown request - make_account";
                }
            } else {
                invalid_attempts = invalid_attempts + 1;
                return "Incorrect Password";
            }
        } else {
            return call_the_cops();
        }
    }

    return dispatch;

}
```

Answer of exercise 3.6

```
let state = 2;

function rand(symbol) {
    if (symbol === "reset") {
        return new_state => {
            state = new_state;
        };
    } else {
        // symbol is "generate"
        state = (state * 1010) % 1101;
        return state;
    }
}
```

Answer of exercise 3.7

```
function make_joint(linked_acc, linked_pw, joint_pw) {
    return (message, input_pw) => {

        // Check authentication for joint account
        if (input_pw !== joint_pw) {
            return x => "Wrong joint account password";
        } else {
```

```
            const access_linked = linked_acc(message, linked_pw);

            // Check authentication for linked account
            if (access_linked(0) === "Incorrect Password") {
                // access_linked(0) does a deposit / withdrawal of 0, in order
                // to test for the "Incorrect Password" message.
                return x => "Wrong linked account password";
            } else {
                // All authentication passed, return accessed account to user
                return access_linked;
            }
        }
    };
}
```

Answer of exercise 3.21

```
function last_pair(items) {
    return is_null(tail(items))
           ? items
           : last_pair(tail(items));
}
```

References

Abelson, Harold, Andrew Berlin, Jacob Katzenelson, William McAllister, Guillermo Rozas, Gerald Jay Sussman, and Jack Wisdom. 1992. The Supercomputer Toolkit: A general framework for special-purpose computing. *International Journal of High-Speed Electronics* 3(3):337-361.

Allen, John. 1978. *Anatomy of Lisp.* New York: McGraw-Hill.

ANSI X3.226-1994. *American National Standard for Information Systems—Programming Language—Common Lisp.*

Appel, Andrew W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25(4):275-279.

Backus, John. 1978. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8):613-641.

Baker, Henry G., Jr. 1978. List processing in real time on a serial computer. *Communications of the ACM* 21(4):280-293.

Batali, John, Neil Mayle, Howard Shrobe, Gerald Jay Sussman, and Daniel Weise. 1982. The Scheme-81 architecture—System and chip. In *Proceedings of the MIT Conference on Advanced Research in VLSI,* edited by Paul Penfield, Jr. Dedham, MA: Artech House.

Borning, Alan. 1977. ThingLab—An object-oriented system for building simulations using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence.*

Borodin, Alan, and Ian Munro. 1975. *The Computational Complexity of Algebraic and Numeric Problems.* New York: American Elsevier.

Chaitin, Gregory J. 1975. Randomness and mathematical proof. *Scientific American* 232(5):47-52.

Church, Alonzo. 1941. *The Calculi of Lambda-Conversion.* Princeton, N.J.: Princeton University Press.

Clark, Keith L. 1978. Negation as failure. In *Logic and Data Bases.* New York: Plenum Press, pp. 293-322.

Clinger, William. 1982. Nondeterministic call by need is neither lazy nor by name. In *Proceedings*

of the ACM Symposium on Lisp and Functional Programming, pp. 226-234.

Clinger, William, and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages,* pp. 155-162.

Colmerauer A., H. Kanoui, R. Pasero, and P. Roussel. 1973. Un système de communication homme-machine en franÇais. Technical report, Groupe Intelligence Artificielle, Université d'Aix Marseille, Luminy.

Cormen, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms.* Cambridge, MA: MIT Press.

Darlington, John, Peter Henderson, and David Turner. 1982. *Functional Programming and Its Applications.* New York: Cambridge University Press.

Dijkstra, Edsger W. 1968a. The structure of the 'THE' multiprogramming system. *Communications of the ACM* 11(5):341-346.

Dijkstra, Edsger W. 1968b. Cooperating sequential processes. In *Programming Languages*, edited by F. Genuys. New York: Academic Press, pp. 43-112.

Dinesman, Howard P. 1968. *Superior Mathematical Puzzles.* New York: Simon and Schuster.

deKleer, Johan, Jon Doyle, Guy Steele, and Gerald J. Sussman. 1977. AMORD: Explicit control of reasoning. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages,* pp. 116-125.

Doyle, Jon. 1979. A truth maintenance system. *Artificial Intelligence* 12:231-272.

Feigenbaum, Edward, and Howard Shrobe. 1993. The Japanese National Fifth Generation Project: Introduction, survey, and evaluation. In *Future Generation Computer Systems,* vol. 9, pp. 105-117.

Feeley, Marc. 1986. Deux approches à l'implantation du language Scheme. Masters thesis, Université de Montréal.

Feeley, Marc and Guy Lapalme. 1987. Using closures for code generation. *Journal of Computer Languages* 12(1):47-66.

Feller, William. 1957. *An Introduction to Probability Theory and Its Applications,* volume 1. New York: John Wiley & Sons.

Fenichel, R., and J. Yochelson. 1969. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM* 12(11):611-612.

Floyd, Robert. 1967. Nondeterministic algorithms. *JACM,* 14(4):636-644.

Forbus, Kenneth D., and Johan deKleer. 1993. *Building Problem Solvers.* Cambridge, MA: MIT Press.

Friedman, Daniel P., and David S. Wise. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming: Third International Colloquium,* edited by S. Michaelson and R. Milner, pp. 257-284.

Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. 1992. *Essentials of Programming Languages.* Cambridge, MA: MIT Press/McGraw-Hill.

Gabriel, Richard P. 1988. The Why of *Y. Lisp Pointers* 2(2):15-25.

Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley.

Gordon, Michael, Robin Milner, and Christopher Wadsworth. 1979. *Edinburgh LCF.* Lecture Notes in Computer Science, volume 78. New York: Springer-Verlag.

Gray, Jim, and Andreas Reuter. 1993. *Transaction Processing: Concepts and Models.* San Mateo, CA: Morgan-Kaufman.

Green, Cordell. 1969. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 219-240.

Green, Cordell, and Bertram Raphael. 1968. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the ACM National Conference,* pp. 169-181.

Griss, Martin L. 1981. Portable Standard Lisp, a brief overview. Utah Symbolic Computation Group Operating Note 58, University of Utah.

Guttag, John V. 1977. Abstract data types and the development of data structures. *Communications of the ACM* 20(6):397-404.

Hamming, Richard W. 1980. *Coding and Information Theory.* Englewood Cliffs, N.J.: Prentice-Hall.

Hanson, Christopher P. 1990. Efficient stack allocation for tail-recursive languages. In *Proceedings of ACM Conference on Lisp and Functional Programming,* pp. 106-118.

Hanson, Christopher P. 1991. A syntactic closures macro facility. *Lisp Pointers,* 4(3).

Hardy, Godfrey H. 1921. Srinivasa Ramanujan. *Proceedings of the London Mathematical Society* XIX(2).

Hardy, Godfrey H., and E. M. Wright. 1960. *An Introduction to the Theory of Numbers.* 4th edition. New York: Oxford University Press.

Havender, J. 1968. Avoiding deadlocks in multi-tasking systems. *IBM Systems Journal* 7(2):74-84.

Hearn, Anthony C. 1969. Standard Lisp. Technical report AIM-90, Artificial Intelligence Project, Stanford University.

Henderson, Peter. 1980. *Functional Programming: Application and Implementation.* Englewood Cliffs, N.J.: Prentice-Hall.

Henderson. Peter. 1982. Functional Geometry. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pp. 179-187.

Hewitt, Carl E. 1969. PLANNER: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 295-301.

Hewitt, Carl E. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* 8(3):323-364.

Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Informatica* 1(1).

Hodges, Andrew. 1983. *Alan Turing: The Enigma.* New York: Simon and Schuster.

Hofstadter, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid.* New York: Basic Books.

Hughes, R. J. M. 1990. Why functional programming matters. In *Research Topics in Functional Programming*, edited by David Turner. Reading, MA: Addison-Wesley, pp. 17-42.

IEEE Std 1178-1990. 1990. *IEEE Standard for the Scheme Programming Language.*

Ingerman, Peter, Edgar Irons, Kirk Sattley, and Wallace Feurzeig; assisted by M. Lind, Herbert Kanner, and Robert Floyd. 1960. THUNKS: A way of compiling procedure statements, with some comments on procedure declarations. Unpublished manuscript. (Also, private communication from Wallace Feurzeig.)

Kaldewaij, Anne. 1990. *Programming: The Derivation of Algorithms.* New York: Prentice-Hall.

Kohlbecker, Eugene Edmund, Jr. 1986. Syntactic extensions in the programming language Lisp. Ph.D. thesis, Indiana University.

Konopasek, Milos, and Sundaresan Jayaraman. 1984. *The TK!Solver Book: A Guide to Problem-Solving in Science, Engineering, Business, and Education.* Berkeley, CA: Osborne/McGraw-Hill.

Knuth, Donald E. 1973. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming.* 2nd edition. Reading, MA: Addison-Wesley.

Knuth, Donald E. 1981. *Seminumerical Algorithms.* Volume 2 of *The Art of Computer Programming.* 2nd edition. Reading, MA: Addison-Wesley.

Kowalski, Robert. 1973. Predicate logic as a programming language. Technical report 70, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh.

Kowalski, Robert. 1979. *Logic for Problem Solving.* New York: North-Holland.

Lamport, Leslie. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558-565.

Lampson, Butler, J. J. Horning, R. London, J. G. Mitchell, and G. K. Popek. 1981. Report on the programming language Euclid. Technical report, Computer Systems Research Group, University of Toronto.

Landin, Peter. 1965. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89-101.

Lieberman, Henry, and Carl E. Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6):419-429.

Liskov, Barbara H., and Stephen N. Zilles. 1975. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering* 1(1):7-19.

McAllester, David Allen. 1978. A three-valued truth-maintenance system. Memo 473, MIT Artificial Intelligence Laboratory.

McAllester, David Allen. 1980. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory.

McCarthy, John. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4):184-195.

McCarthy, John. 1967. A basis for a mathematical theory of computation. In *Computer Programing and Formal Systems*, edited by P. Braffort and D. Hirschberg. North-Holland.

McCarthy, John. 1978. The history of Lisp. In *Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages.*

McCarthy, John, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. 1965. *Lisp 1.5 Programmer's Manual.* 2nd edition. Cambridge, MA: MIT Press.

McDermott, Drew, and Gerald Jay Sussman. 1972. Conniver reference manual. Memo 259, MIT Artificial Intelligence Laboratory.

Miller, Gary L. 1976. Riemann's Hypothesis and tests for primality. *Journal of Computer and System Sciences* 13(3):300-317.

Miller, James S., and Guillermo J. Rozas. 1994. Garbage collection is fast, but a stack is faster. Memo 1462, MIT Artificial Intelligence Laboratory.

Moon, David. 1978. MacLisp reference manual, Version 0. Technical report, MIT Laboratory for Computer Science.

Moon, David, and Daniel Weinreb. 1981. Lisp machine manual. Technical report, MIT Artificial Intelligence Laboratory.

Morris, J. H., Eric Schmidt, and Philip Wadler. 1980. Experience with an applicative string processing language. In *Proceedings of the 7th Annual ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages.*

Phillips, Hubert. 1934. *The Sphinx Problem Book.* London: Faber and Faber.

Pitman, Kent. 1983. The revised MacLisp Manual (Saturday evening edition). Technical report 295, MIT Laboratory for Computer Science.

Rabin, Michael O. 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory* 12:128-138.

Raymond, Eric. 1993. *The New Hacker's Dictionary.* 2nd edition. Cambridge, MA: MIT Press.

Raynal, Michel. 1986. *Algorithms for Mutual Exclusion.* Cambridge, MA: MIT Press.

Rees, Jonathan A., and Norman I. Adams IV. 1982. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming,* pp. 114-122.

Rees, Jonathan, and William Clinger (eds). 1991. The revised[4] report on the algorithmic language Scheme. *Lisp Pointers,* 4(3).

Rivest, Ronald, Adi Shamir, and Leonard Adleman. 1977. A method for obtaining digital signatures and public-key cryptosystems. Technical memo LCS/TM82, MIT Laboratory for Computer Science.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12(1):23.

Robinson, J. A. 1983. Logic programming—Past, present, and future. *New Generation Computing* 1:107-124.

Sagade, Y. 2015. SICP exercise 1.14

Spafford, Eugene H. 1989. The Internet Worm: Crisis and aftermath. *Communications of the ACM* 32(6):678-688.

Steele, Guy Lewis, Jr. 1977. Debunking the 'expensive procedure call' myth. In *Proceedings of the National Conference of the ACM,* pp. 153-62.

Steele, Guy Lewis, Jr. 1982. An overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming,* pp. 98-107.

Steele, Guy Lewis, Jr. 1990. *Common Lisp: The Language.* 2nd edition. Digital Press.

Steele, Guy Lewis, Jr., and Gerald Jay Sussman. 1975. Scheme: An interpreter for the extended lambda calculus. Memo 349, MIT Artificial Intelligence Laboratory.

Steele, Guy Lewis, Jr., Donald R. Woods, Raphael A. Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow. 1983. *The Hacker's Dictionary.* New York: Harper & Row.

Stoy, Joseph E. 1977. *Denotational Semantics.* Cambridge, MA: MIT Press.

Sussman, Gerald Jay, and Richard M. Stallman. 1975. Heuristic techniques in computer-aided

circuit analysis. *IEEE Transactions on Circuits and Systems* CAS-22(11):857-865.

Sussman, Gerald Jay, and Guy Lewis Steele Jr. 1980. Constraints—A language for expressing almost-hierarchical descriptions. *AI Journal* 14:1-39.

Sussman, Gerald Jay, and Jack Wisdom. 1992. Chaotic evolution of the solar system. *Science* 257:256-262.

Sussman, Gerald Jay, Terry Winograd, and Eugene Charniak. 1971. Microplanner reference manual. Memo 203A, MIT Artificial Intelligence Laboratory.

Sutherland, Ivan E. 1963. SKETCHPAD: A man-machine graphical communication system. Technical report 296, MIT Lincoln Laboratory.

Teitelman, Warren. 1974. Interlisp reference manual. Technical report, Xerox Palo Alto Research Center.

Thatcher, James W., Eric G. Wagner, and Jesse B. Wright. 1978. Data type specification: Parameterization and the power of specification techniques. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 119-132. Turner, David. 1981. The future of applicative languages. In *Proceedings of the 3rd European Conference on Informatics,* Lecture Notes in Computer Science, volume 123. New York: Springer-Verlag, pp. 334-348.

Wand, Mitchell. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):164-180.

Waters, Richard C. 1979. A method for analyzing loop programs. *IEEE Transactions on Software Engineering* 5(3):237-247.

Winograd, Terry. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report AI TR-17, MIT Artificial Intelligence Laboratory.

Winston, Patrick. 1992. *Artificial Intelligence.* 3rd edition. Reading, MA: Addison-Wesley.

Zabih, Ramin, David McAllester, and David Chapman. 1987. Non-deterministic Lisp with dependency-directed backtracking. *AAAI-87*, pp. 59-64.

Zippel, Richard. 1979. Probabilistic algorithms for sparse polynomials. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT.

Zippel, Richard. 1993. *Effective Polynomial Computation.* Boston, MA: Kluwer Academic Publishers.

Index

> Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.
> — Donald E. Knuth, Fundamental Algorithms (Volume 1 of The Art of Computer Programming)

Page numbers for code definitions are in italics.
Page numbers followed by *n* indicate footnotes.

JavaScript Adaptation Making-of

Like the Source Academy, the JavaScript adaptation of SICP is an open-source community effort. The software and data required for making these web pages and the PDF edition are contained in the repository Source Academy / sicp, and improvements, extensions and discussions are handled in this repository as with many other open-source software projects.

Martin Henz started translating SICP to JavaScript in 2008. He obtained the original LaTeX sources of the textbook from the Gerald Sussman, and converted it to an XML format that allowed him to retain the original sources along with the JavaScript adaptation in a single file. He developed a processing system to generate HTML from XML, using XSLT, resulting in the first version of the JavaScript adaptation.

The mobile-friendly web edition of SICP JS was designed and implemented by Liu Hang in 2017 and then further developed by Feng Piaopiao in 2018. Liu Hang decided to use Ruby on Rails for generating the HTML pages from the XML sources. The XML documents are processed using Nokogiri. For that, the website is originally hosted as a Ruby on Rails application and the generated HTML files are then collected as a pure-HTML5 website. Formulas are retained in the resulting HTML files and are type-set by the reader's browser on-the-fly, using the MathJax system.

In the textbook, program fragments often require other program fragments. In order to collect and execute the necessary programs, the corresponding `SNIPPET` tags in the xml files include `REQUIRES` tags. The Rails server uses these tags in order to assemble the executable programs.

The PDF edition of SICP JS was designed and implemented by Chan Ger Hean in 2019. Ger Hean decided to use Node.js for generating files from the XML sources. The files are then typeset using the PdfLaTeX system.

The e-book edition of SICP JS was designed and implemented by Jolyn Tan in 2019. Jolyn decided to use Node.js for generating files from the XML sources. The files are then processed into the EPUB 3 format using the pandoc system.

The figures are adapted from HTML5/EPUB3 version of SICP by Andres Raba. The figures are licensed under Creative Commons Attribution-ShareAlike 4.0 International License (cc by-sa). JavaScript adaptations of figures were done by manually by Tobias Wrigstad using Inkscape

and gratuitous use of sed.