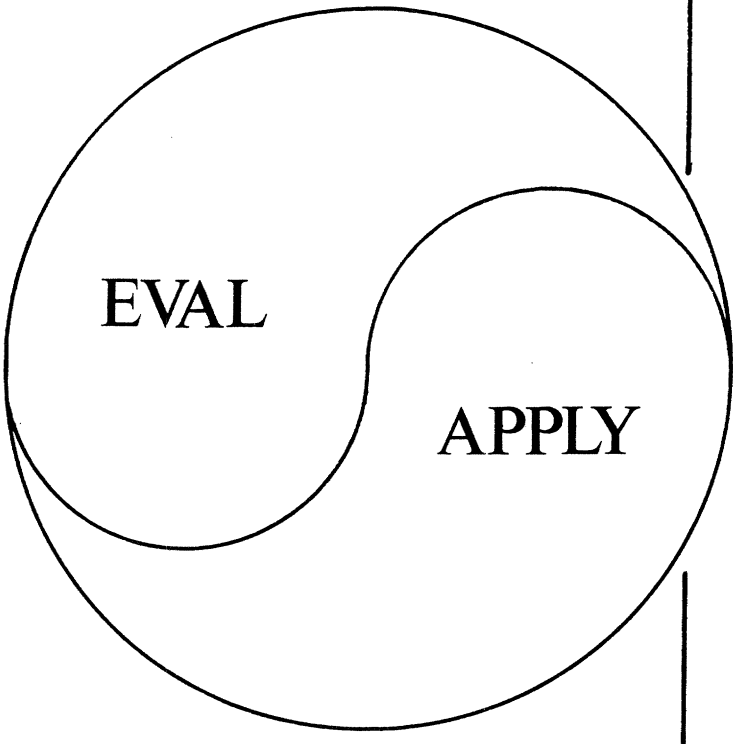
Technical Report 735

Structure & Interpretation of Computer Programs

Harold Abelson
Gerald Jay Sussman



MIT Artificial Intelligence Laboratory

Table of Contents

List of Exercises

Chapter 1: Building Abstractions with Procedures

Chapter 2: Building Abstractions with Data

Chapter 3: Modularity, Objects, and State

Chapter 4: Meta-Linguistic Abstraction

Chapter 5: Register Machine Model of Control

Structure and Interpretation of Computer Programs

Harold Abelson
Gerald Jay Sussman

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

DRAFT: 31 JULY 1983

This is a preliminary draft of a book to be published in the spring of 1984 by the MIT Press and McGraw-Hill. Any comments and suggestions will be greatly appreciated. Please send them to HAL@MIT-MC or GJS@MIT-OZ, or mail them to Harold Abelson or Gerald Jay Sussman, 545 Technology Square, MIT, Cambridge, Ma.

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

-- Alan Perlis

This book is dedicated, in respect and admiration:

To the spirit that lives in the computer.

Prologue

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 Institutue 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 approximately 600 students each year. Most of these students have had little or no prior formal training in computation, although most 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. Secondly, 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 a student who completes this subject should have a good feel for the elements of style and the aesthetics of programming. He should also have command of the major techniques for controlling complexity in a large system. He should be capable of reading a 50 page long program, if it is written in an exemplary style. He should know what not to read, and what he need not understand at any moment. Thus he 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 de-emphasizes others.

Underlying our approach to this subject is our conviction that "Computer Science" is not a science, and that its significance has has little to do with computers. The computer revolution is a revolution in the way we think, and in the way in which 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 precisely dealing with notions of "what is". Computation provides a framework for precisely dealing 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 covered in an hour, like the rules of chess. After a short time we forget about 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 high-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 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 of 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-60. From Lisp we take the meta-linguistic power, deriving from the simple syntax, the uniform representation of programs as data objects, and the garbage-collected heap-allocated data. From Algol we take the lexical scoping rule, and block structure, which are gifts from the pioneers of programming language design who were on the Algol-60 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. Our heroes include Alonzo Church, Barkley Rosser, Stephen Kleene, and Haskell Curry.

Acknowledgements

We would like to thank all of the 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 in the lambda calculus, taught by Jack Wozencraft and Arthur Evans, Jr. at MIT in the late 1960's.

We owe a great debt to Robert Fano, who reorganized the introductory curriculum around 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 which we try to teach in 6.001 were

developed in conjunction with Guy Lewis Steele Jr., who collaborated with Sussman in the initial development of the Scheme language.

Joel Moses taught us about structuring large systems. His experience with the Macsyma system 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 which 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.

To this we added an attitude from 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 artists at work. We are fortunate to have programmed at the feet of Bill Gosper and Richard Greenblatt.

It is difficult to identify all of the people who have made particular technical contributions to the development of our curriculum. We thank all the recitation instructors and tutors who have worked with us over the past few years, especially Bill Siebert, Albert Meyer, Joe Stoy, and Randy Davis, who put in many extra hours on our subject. Julie Sussman has painstakingly read many drafts of the manuscript, working out every exercise, and catching many bugs. (Unfortunately, we have introduced new bugs in later drafts.) Julie's incisive comments and clear anticipation of student problems has been a major help to us in organizing this presentation. David Turner, Peter Henderson, Dan Freidman, David Wise, and Will Clinger have taught us many of the techniques of the functional programming community that appear in this book.

It is also hard to enumerate all of the people who have made particular 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, and Guillermo Rozas. Others who have put in significant time are been: Richard Stallman, Alan Bawden, Kent Pitman, Jon Taft, Neil Mayle, Jon Lamping, Gwyn Osnos, Tracy Larabee, George Carrette, Soma Chaudhuri, Bill Chiarchiaro, Steven Kirsch, Leigh Klotz, Wayne Noss, Patrick O'Donnell, Kevin Theobald, Daniel Weise, Kenneth Sinclair, Anthony Courtmanche, and Andrew Berlin.

We are pleased that others are working on similar language implementations and we hope that we will continue to learn from each other's activity. We want especially to draw attention to the work on "T" by Jon Rees Kent Pitman and others at Yale, and the work on "Scheme-311" by Mitch Wand, Will Clinger, Dan Friedman, and others at the University of Indiana.

Finally, we would like to thank all of the people who have supported and encouraged this work, including Ira Goldstein and Joel Birnbaum at Hewlett-Packard Laboratories and Bob Kahn at DARPA.

Chapter 1
Building Abstractions with Procedures

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

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

We are about to study the idea of a *computational process*. Computational processes are abstract, almost magical beings that inhabit computers. Processes manipulate other abstract things called *data*. In their tasks, processes are directed by sets of rules or patterns called *programs*. Human engineers create programs to direct processes. In effect, we conjure spirits of the computer with our spells.

A computational process is indeed very like the 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 very much like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks that we want our processes to perform.

A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the proverbial sorcerer's apprentice, the problem of the novice programmer is to understand and to anticipate the consequences of his 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.

A master software engineer has the ability to organize programs so that he can be reasonably sure his processes will perform the tasks intended. He can *previsualize* the behavior of his system. He knows how to structure his programs so that unanticipated problems do not lead to catastrophic consequences, and when problems do arise, he can *debug* his 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 Lisp

We need an appropriate language for describing processes, and we will use for this purpose the programming language *Lisp*. Just as our everyday thoughts are usually expressed in our natural language (such as English, or French, or Japanese), and descriptions of quantitative phenomena are expressed with mathematical notations, our procedural thoughts will be expressed in Lisp. Lisp was invented in the late 1950's as a formalism for reasoning about the use of certain kinds of logical expressions, called *recursion equations*, as a model for computation. The language was conceived by John McCarthy and is based on his paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine" [30].

Despite its inception as a mathematical formalism, Lisp is a practical programming language. A Lisp *interpreter* is a machine (commonly implemented as a program that makes a commercial computer simulate a machine) that carries out processes described in the Lisp language. The first Lisp interpreter was implemented by McCarthy with the help of colleagues and students in the Artificial Intelligence Group of the MIT Research Laboratory of Electronics and in the MIT Computation Center.[1] Lisp, whose name is an acronym for LISt Processing, was designed to provide symbol manipulating capabilities for attacking programming problems such as the symbolic differentiation and integration of algebraic expressions. It included for this purpose new data objects known as atoms and lists, which most strikingly set it apart from all other languages of the period.

Lisp was not the product of a concerted design effort. Instead, it evolved informally in an experimental manner in response to user needs and pragmatic implementation considerations. Lisp's informal evolution has continued through the years, and the community of Lisp users has traditionally resisted attempts to promulgate any "official" definition of the language. This evolution, together with the flexibility and elegance of the initial conception, has enabled Lisp, which is the second oldest language in widespread use today (only Fortran is older), to continually adapt to encompass the most modern ideas about program design. Thus Lisp is by now a family of dialects, which, while sharing most of the original features, may differ from one another in significant ways. The dialect of Lisp used in

[1] The "Lisp 1 Programmer's Manual" appeared in 1960, and the "Lisp 1.5 Programmer's Manual" [31] was published in 1962. The early history of Lisp is described by McCarthy in [32].

this book is called Scheme.[2]

Because of its experimental character and its emphasis on symbol manipulation, Lisp was originally inordinately inefficient for numerical computations, at least when compared to Fortran. Over the past fifteen years, however, Lisp compilers have been developed that translate programs into machine code that can perform numerical computations as efficiently as code generated from any other high-level language. In spite of this, Lisp has not yet overcome its old reputation as a hopelessly inefficient language, and its use is still localized in a few research laboratories.

If Lisp is not a popular language, why are we using it as the framework for our discussion of programming? Because the language possesses unique 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. The most significant of these features is the fact that Lisp descriptions of processes, called *procedures*, can themselves be represented and manipulated as Lisp data. The importance of this is that there are powerful program design techniques that rely on the ability to blur the traditional distinction between "passive" data and "active" processes. As we shall discover, Lisp's flexibility in handling procedures as data makes it one of the most convenient languages in existence for exploring these techniques. The ability to represent procedures as data also makes Lisp an excellent language for writing programs that must manipulate other programs as data, such as the interpreters and compilers that support computer languages. Above and beyond these considerations, programming in Lisp is great fun.

1.1. The Elements of Programming

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

- There are *primitive expressions* that represent the simplest entities with which the language is concerned.
- There are *means of combination* by which compound expressions are built from simpler ones.

---

[2] The two dialects in which most major Lisp programs of the 1970's were written are MacLisp [33], developed at the MIT Project MAC, and Interlisp [48], developed at Bolt Beranek and Newman and the Xerox Palo Alto Research Center. *Portable Standard Lisp* [18, 14] is another Lisp dialect designed to be easily portable between different machines, and is beginning to become widely available. MacLisp has also spawned a number of sub-dialects, such as *Franz Lisp*, which was developed at the University of California at Berkeley, and *Lisp Machine Lisp* [34], which is based on a special-purpose processor designed at the MIT Artificial Intelligence Laboratory to run Lisp very efficiently. *Common Lisp* is another Lisp dialect currently under development, which is meant to serve as a standard for future production Lisp systems [41]. The Lisp dialect used in this book is called Scheme. It was invented in 1975 by Guy Lewis Steele Jr. and Gerald Jay Sussman of the MIT Artificial Intelligence Laboratory [39] and later reimplemented for instructional use at MIT.

- There are *means of abstraction* by which compound objects can be named and manipulated as units.

In programming, there are two kinds of objects with which we deal: procedures and data (though we will discover that they are really not so distinct). Informally, data is "stuff" that represents specific objects that we want to manipulate, and procedures are descriptions of the rules for manipulating the data. Thus any powerful programming language should be able to describe primitive data and primitive procedures, and should have methods for combining and abstracting procedures and data.

In this chapter we will deal only with simple numerical data so that we can focus on the rules for building procedures.[3] In later chapters we will see that these same rules allow us to build procedures to manipulate compound data as well.

1.1.1. Expressions

One easy way to get started programming is to examine some typical interactions with an interpreter for the Scheme dialect of Lisp. Imagine that we are sitting at a computer terminal, and that the interpreter has indicated that it is ready to serve us by displaying a *prompt*

```
==>
```

at the beginning of a blank line. If we respond to the prompt by typing an expression, the interpreter responds by displaying the result of *evaluating* that expression.

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

```
==> 486
```

the interpreter will respond by printing[4]

486

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

---

[3] The characterization of numbers as "simple data" is a barefaced bluff. In fact, the treatment of numbers is one of the trickiest and most confusing aspects of any programming language. Some typical issues involved are: Is there a difference between integers, such as 2, and "real" numbers, such as 2.00? Are the arithmetic operators used for integers the same as the operators used for real numbers? Does 6 divided by 2 produce 3 or 3.0? How large a number can we represent? How many decimal places of accuracy can we represent? Is the range of integers the same as the range of real numbers? Above and beyond these questions, of course, lies a collection of issues concerning roundoff and truncation errors -- the entire science of numerical analysis. Since our focus in this book is on large-scale program design rather than on numerical techniques, we are going to ignore these problems. The Scheme dialect of Lisp, wherever possible, does not distinguish between integers and "real" numbers (for example, 3 is equal to 3.0). The numerical examples in this chapter will exhibit the usual roundoff behavior that one observes when using arithmetic operations that preserve about 7 decimal places of accuracy in non-integer operations.

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

```
==> (+ 137 349)
486

==> (- 1000 334)
666

==> (* 5 99)
495

==> (/ 10 5)
2

==> (/ 10 6)
1.66667

==> (+ 2.7 10)
12.7
```

Expressions such as these, formed by delimiting a list of expressions within parentheses, are called *combinations*. The leftmost element in the list is called the *operator* and the other elements are called *operands*. The value of a combination is obtained by applying the procedure specified by the operator to the arguments which are the values of the operands.

The convention of placing the operator to the left of its arguments is known as *prefix notation*, and it may be somewhat confusing at first because it departs significantly from the mathematical convention to which we are accustomed. Prefix notation has several advantages, however. One of them is that the notation can accommodate operators that may take an arbitrary number of arguments, as in the following examples:

```
==> (+ 21 35 12 7)
75

==> (* 25 4 12)
1200
```

No ambiguity can arise, because the operator is always the leftmost element and the entire combination is delimited by the parentheses.

A second advantage of prefix notation is that it extends in a straightforward way to allow combinations to be *nested*, that is, to have combinations whose elements are themselves combinations:

```
==> (+ (* 3 5) (- 10 6))
19
```

There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions that the Lisp interpreter can evaluate. It is we humans who get confused by still relatively simple expressions such as

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

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

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

following a formatting convention known as *pretty-printing*, in which each long combination is written so that the operands are aligned vertically. The resulting indentations display clearly the structure of the expression.[5]

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

1.1.2. Naming and the Environment

A critical aspect of a programming language is the means it provides for allowing one to use names to refer to computational objects. We say that the name identifies a *variable* whose *value* is the object.

In the Scheme dialect of Lisp, the operator for naming things is called `define`. Typing

```
==> (define size 2)
size
```

causes the interpreter to associate the value 2 with the name `size`. Notice that the interpreter responds to a `define` combination by printing the name being defined.[6]

Once the name `size` has been defined to be the number 2, we can refer to the value 2 by name:

```
==> size
2

==> (* 5 size)
10
```

---

[5] Lisp systems typically provide features to aid the user in formatting expressions. Two especially useful features are to automatically indent to the proper pretty-print position whenever a new line is started, and to highlight the matching left parenthesis whenever a right parenthesis is typed.

[6] The symbol printed is actually the *value* of the `define` combination. In Lisp, one makes the convention that *every expression has a value*. This requirement may seem silly, but deviating from it would cause more bothersome complications. It also meshes nicely with the read-eval-print mode in which the interpreter operates, since it ensures that the interpreter will have something to print in response to evaluating any expression. When there is no natural choice for the value to be returned as the result of an operation, language implementors choose a value by convention, as in the case of `define`. The conventions for choosing such values tend to be highly implementation dependent, and it is dangerous practice to write programs that rely on them. (The convention that every Lisp expression have a value, together with the old reputation of Lisp as an inefficient language, is the source of the quip by Alan Perlis that "Lisp programmers know the value of everything but the cost of nothing.")

Here are further examples of the use of *define*:

```
==> (define pi 3.14159)
pi

==> (define radius 10)
radius

==> (* pi (* radius radius))
314.159

==> (define circumference (* 2 pi radius))
circumference

==> circumference
62.8318
```

Define, as the basic mechanism for naming, is our language's simplest means of *abstraction*. Computational objects may have very complex structures, and it would be extremely inconvenient to have to remember and repeat their details each time we want to use them. Indeed, complex programs are constructed by building, step by step, computational objects of increasing complexity. The interpreter makes this step-by-step program construction particularly convenient because name-object associations can be created incrementally in successive interactions. This feature encourages the incremental development and testing of programs, and is largely responsible for the fact that Lisp programs usually consist of a large number of relatively simple procedures.

It should be clear that the possibility of associating values with symbols and later retrieving them means that the interpreter must maintain some sort of memory that keeps track of the name-object pairs. This memory is called the *environment*, or more precisely the *global environment*, since we will see later that a computation may involve a number of different environments.[7]

1.1.3. Evaluating Combinations

One of our goals in this chapter is to isolate issues about thinking procedurally. As a case in point, let us consider that, in evaluating combinations, the Lisp interpreter is itself following a procedure. For the expressions we have discussed so far the evaluation process is simply described.

To evaluate a combination (other than a definition):

1. Evaluate the subexpressions of the combination.
2. Apply the procedure which is the value of the leftmost subexpression (the operator) to the arguments which are the values of the other subexpressions (the operands).

---

[7] In Chapter 3, we shall see that this notion of environment is crucial, both for understanding how the interpreter works and for implementing interpreters.

Even this simple rule illustrates some important points about processes in general. First, observe that step 1 dictates that in order to accomplish the evaluation process for a combination we must first perform the evaluation process on each element of the combination. Thus the evaluation rule is *recursive* in nature; that is, it includes, as one of its steps, the need to invoke the rule itself.[8]

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

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

requires that the evaluation rule be applied to four different combinations. We can obtain a picture of this process by representing the combination in the form of a tree, as shown in figure 1-1. Each combination is represented by a node, from which stem branches corresponding to the operator and operands of the combination. The terminal nodes (that is, nodes with no branches stemming from them) represent either operators or numbers. Viewing evaluation in terms of the tree, we can imagine that the values of the operands percolate upwards, starting from the terminal nodes and then combining at higher and higher levels. In general, we shall see that recursion is a very powerful technique for dealing with hierarchical, tree-like objects. In fact, the "percolate values upwards" form of the evaluation rule is an example of a general kind of process known as *tree accumulation*.

[8] It may seem strange that the evaluation rule includes, as part of Step 1, that we should evaluate the leftmost element of a combination which, so far as we have seen, can only be an operator representing a built-in primitive procedure such as + or *. We will see later on that it is in fact useful to be able to work with combinations whose operators are themselves compound expressions.

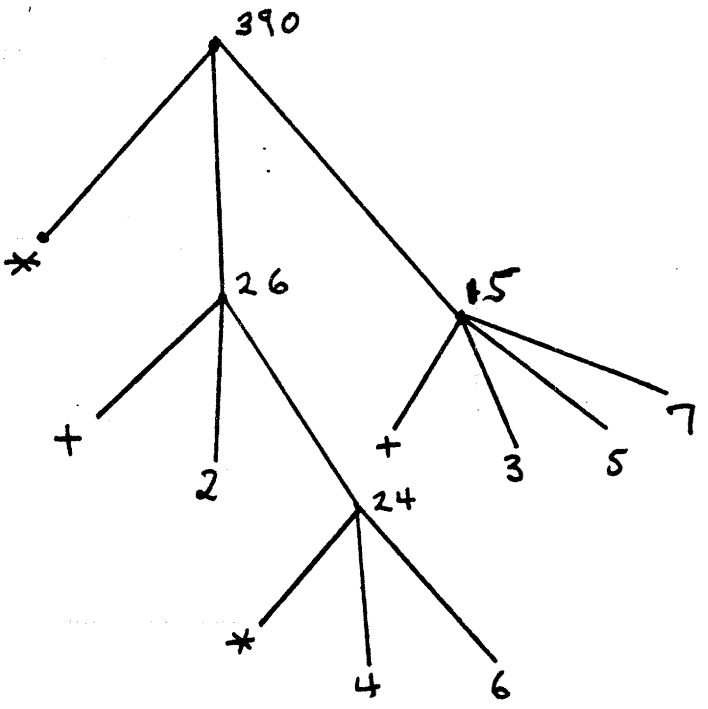


Figure 1-1: Tree representation, showing the value of each subcombination.

Next, observe that the repeated application of step 1 brings us to the point where we need to evaluate, not combinations, but primitive expressions such as numerals, built-in operators, or other names. We take care of the primitive cases by stipulating that:

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

We may regard the second rule as a special case of the third one, by imagining that symbols such as + and * are also included in the global environment, associated with the sequences of machine instructions that are their "values." The key point to notice is the role of the environment in determining the *meaning* of the symbols in the expressions. In an interactive language such as Lisp, it is in a certain sense meaningless to ask for the value of an expression such as

```
(+ x 1)
```

without specifying any information about the environment that would provide a meaning for the symbol "x" (or even for the symbol "+"!). As we shall see in Chapter 3, the general notion of the environment as providing a *context* in which evaluation takes place will play an important role in our understanding of program execution.

Finally, notice that *define* is an exception to the general evaluation rule given above. For instance, evaluating the expression

```
(define x 3)
```

does not apply *define* to two arguments, one of which is the value of the symbol *x* and the other of which is 3, since the purpose of the *define* is precisely to associate *x* with a value.

Such exceptions to the general evaluation rule are called *special forms*. *Define* is the only example of a special form which we have seen so far, but we shall meet others shortly. Each special form has its own way in which the general evaluation rule should be modified in order to handle it. The special forms and their associated special evaluation rules constitute the syntax of the programming language. In comparison to most other programming languages, Lisp has a very simple syntax; that is, the evaluation rule for expressions can be described by a simple general rule together with specialized rules for a small number of special forms.[9]

1.1.4. Compound procedures

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

- Numbers and arithmetic operators are primitive data and procedures.
- Nesting of combinations provides a means of combining operators.
- Using *define* to associate names with values provides a limited means of abstraction.

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

We begin by examining how to express the idea of "squaring." We might say, "To square something, multiply it by itself." This is expressed in our language as follows:

```
(define (square x) (* x x))
```

We can understand this in the following way:

```
(define (square x)        (*        x     x))
  |       |    |          |        |     |
  |       |    |          |        |     |
 To    square something, multiply it by itself.
```

We have here a *compound procedure*, which has been given the name *square*. It represents the operation of multiplication of an entity by itself. The entity to be multiplied is

---

[9] Special syntactic forms that are simply convenient alternative surface structure for things that can be written in more uniform ways, are sometimes called *syntactic sugar*, to use a phrase coined by Peter Landin. When compared to users of other languages, Lisp programmers, as a rule, tend to be unconcerned with matters of syntax. (By contrast, examine any Pascal manual, and notice how much of it is devoted to descriptions of syntax.) This disdain for syntax is partially due to the flexibility of Lisp, which makes it easy to change surface syntax, and partly due to the observation that many "convenient" syntactic constructs, which make the language less uniform, end up causing more trouble than they are worth when programs become large and complex. In the words of Alan Perlis: "Syntactic sugar causes cancer of the semicolon."

given a local name, *x*, which plays the same role that a pronoun plays in natural language.

Executing the *define* form causes the specified procedure name to be associated with the corresponding procedure definition in the environment. The interpreter responds to *define* by printing the name of the procedure being defined:

```
==> (define (square x) (* x x))
square
```

The general form of a procedure definition is

(define (*⟨name⟩ ⟨formal parameters⟩*) *⟨body⟩*)

The *⟨name⟩* is a symbol to be associated with the procedure definition in the environment.[10] The *⟨formal parameters⟩* are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The *⟨body⟩* is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied.[11] Observe that the *⟨name⟩* and the *⟨formal parameters⟩* are grouped within parentheses, just as they would in an actual call to the procedure being defined.

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

```
==> (square 21)
441

==> (square (+ 2 5))
49

==> (square (square 3))
81
```

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

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

We can easily define a procedure *sum-of-squares* which, given any two numbers as arguments, produces the sum of their squares:

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

And if we were to define the following procedure:

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

---

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

[11] More generally, the body of the procedure can be a sequence of combinations. In this case, the interpreter evaluates each combination in the sequence in turn, and returns the value of the final combination as the value of the procedure application.

we would have that *(f 5)* is $6^2 + 10^2$, or 136. Notice that defined procedures are used in *exactly* the same way as primitive procedures. Indeed, one could not tell by looking at the definition of *sum-of-squares* given above whether *square* was built into the interpreter or defined as a compound procedure.

1.1.5. The Substitution Model for Procedure Evaluation

To evaluate a combination whose operator is a compound procedure, the interpreter follows much the same process as for combinations whose operators are primitive procedures, as we discussed in section 1.1.3. That is, the interpreter evaluates the elements of the combination and applies the procedure (which is the value of the operator of the combination) to the arguments (which are the values of the operands of the combination).

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

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

To illustrate this process, let's evaluate the combination

```
(f 5)
```

where *f* is the procedure defined in section 1.1.4 above. We begin by retrieving the body of *f*

```
(sum-of-squares (+ a 1) (* a 2)))
```

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

```
(sum-of-squares (+ 5 1) (* 5 2))
```

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

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

Using the definition of *square*, this reduces to

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

which reduces by multiplication to

```
(+ 36 100)
```

and finally to

```
136
```

The process we have just described is called the *substitution model* for procedure evaluation, and it can be taken as a model that determines the "meaning" of procedure application, insofar as the procedures in this chapter are concerned. However, there are two points that should be stressed:

1. The substitution model is a *model* that allows one to think about procedure application. Typical interpreters do *not* evaluate procedure applications by operating on the text of a procedure to substitute values for the formal parameters. In practice, the "substitution" is accomplished by using a *local environment* for the formal parameters. We will discuss this more fully in Chapters 3 and 4 we will examine the implementation of an interpreter in detail.
2. The substitution model is not powerful enough to describe all of the procedures we will consider in this book. In particular, when we address in Chapter 3 the use of procedures with so-called "mutable data," we will see that the substitution model breaks down and must be replaced by a more complicated model of procedure application. On the other hand, substitution is a straightforward idea. It serves well for understanding all of the procedures in the first two chapters of this book and indeed, for understanding most of the procedures one normally encounters.[12] The model is a good tool to use, so long as we bear in mind that it does have limitations.

Notice that, according to the model given above, the interpreter *first* evaluates the arguments to a procedure, and *then* applies the procedure to the evaluated arguments. This is not the only way to perform evaluation. An alternative substitution model would first expand each procedure definition in terms of simpler and simpler procedures, until we obtain an expression involving only primitive operators, and then perform the evaluation. If we used this method, then the evaluation of

```
(f 5)
```

would proceed according to the following sequence of expansions:

```
(sum-of-squares (+ 5 1) (* 5 2))

(+    (square (+ 5 1))      (square (* 5 2))  )

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

followed by the reductions

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

(+            36                   100)

                    136
```

This gives the same answer as our previous substitution model, but the *process* is different.

---

[12] Despite the fact that substitution is a "straightforward idea" it turns out to be surprisingly complicated to give a rigorous mathematical definition of the substitution process. The problem arises from the possibility of confusion between the names used for the formal parameters of a procedure and the (possibly identical) names used in the expressions to which the procedure may be applied. Indeed, there is a long history of erroneous definitions of "substitution" in the literature of logic and programming semantics. See the book by Joseph Stoy [43] for a careful discussion of substitution. And yet, from a formal mathematical perspective, substitution is much simpler to contend with rigorously than the more complete interpreter model that we shall discuss in later chapters, which, at the current state of the art, seems hardly mathematically tractable at all.

Notice in particular that the evaluations of *(+ 5 1)* and *(* 5 2)* are each performed twice here, corresponding to the reduction of the expression

```
(* x x)
```

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

This alternative "fully expand and then reduce" evaluation method is known as *normal order evaluation*, in contrast to the "evaluate the arguments and then apply" method that the interpreter actually uses, which is called *applicative order evaluation*. It can be shown that, for procedure applications that can be modeled using substitution (including all the procedures in the first two chapters of this book) and that yield legitimate values, normal and applicative order evaluation produce the same value. (See exercise 1-4 for an instance of an "illegitimate" value where normal and applicative order evaluation would not give the same result.) Most interpreters use applicative order evaluation, partly because of the additional efficiency obtained from avoiding the kind of multiple evaluations of expressions illustrated with *(+ 5 1)* and *(* 5 2)* above, and, more significantly, because normal order evaluation becomes much more complicated to deal with when we leave the realm of procedures that can be modeled by substitution, as we will do in Chapter 3. On the other hand, normal order evaluation can also be a useful technique. When we tackle the problem of coping with "infinite data structures," we will use a method closely akin to normal order evaluation.[13]

1.1.6. Conditional Expressions and Predicates

The expressive power of the class of procedures that we can define at this point is very limited. For instance, we cannot define a procedure that computes the absolute value of a number by testing whether the number is positive, negative, or zero and taking different actions in the different cases according to the rule:

$$\text{abs}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

This construct is called a *case analysis* and there is a special form in Lisp for notating such a case analysis. It is called *cond* (which stands for "conditional") and it is used as follows:

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

The general form of a conditional expression is

[13] In Chapter 3 we will introduce the notion of *delayed evaluation* to provide various "intermediate grounds" between normal and applicative orders. We will also introduce *call-by-need* evaluation as a general technique for avoiding the multiple evaluations used in strict normal order evaluation. See Chapter 3, section 3.4.3.

```
(cond (<p1> <e1>)
      (<p2> <e2>)
      .
      .
      .
      (<pn> <en>))
```

in which the arguments are pairs of expressions (*<p> <e>*) called *clauses*. The first expression in each pair is a *predicate* -- that is, an expression whose value is interpreted as either true or false. In Lisp, "false" is represented by the value of the distinguished symbol *nil*, and any other value is interpreted as "true." The symbol *t* is often used as a canonical non-*nil* symbol to represent "true."

Conditional expressions are evaluated as follows. The predicate *<p1>* is evaluated first. If its value is false (*i.e.*, *nil*) then *<p2>* is evaluated. If its value is also false then *<p3>* is evaluated. This process continues until a predicate is found whose value is non-*nil*, in which case the interpreter returns the value of the corresponding *<e>* of the clause as the value of the conditional expression. If none of the *<p>*'s is found to be true, the *cond* returns *nil*.

The *abs* procedure above makes use of the primitive procedures >, <, and =.[14] These are operators that take two numbers as arguments and return *t* if the first number is greater than, less than, or equal to the second number, respectively, and *nil* otherwise.

Another way to write the absolute value procedure is:

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

which could be expressed in English as "if *x* is less than zero return *-x*; otherwise return *x*." *Else* is a special symbol that can be used in place of the *<p>* in the final clause of a *cond*. This causes the *cond* to return as its value the value of the corresponding *<e>* whenever all previous clauses have been bypassed. In fact, any expression that always evaluates to a non-*nil* value could be used here.

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

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

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

(if *<predicate> <consequent> <alternative>*)

To evaluate an *if* expression, the interpreter first evaluates the *<predicate>* part of the expression. If it is non-*nil* the interpreter then evaluates and returns the value of the

---

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

⟨consequent⟩. Otherwise it evaluates and returns the value of the *⟨alternative⟩*.[15]

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

and	Takes an arbitrary number of arguments. If none of the arguments evaluates to *nil*, the value of the *and* is non-*nil*.
or	Takes an arbitrary number of arguments. If all of the arguments evaluate to *nil*, the value of the *or* is *nil*, otherwise it is non-*nil*.
not	Takes a single argument. It returns non-*nil* when the argument evaluates to *nil* and *nil* otherwise.

For instance, the condition that a number x be in the range 5 < x < 10 may be expressed as

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

As another example, we can define a predicate to test whether one number is greater than or equal to another as

```
(define (>= x y)
  (or (> x y) (= x y)))
```

or, alternatively, as

```
(define (>= x y)
  (not (< x y)))
```

Observe that *and* and *or* are special forms, because the interpreter will not necessarily evaluate all the arguments to these operators. It only evaluates as many arguments as are required to determine the value to be returned.

Exercise 1-1: Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? You should assume that the sequence is to be evaluated in the order it is presented.

```
==> 10

==> (+ 5 3 4)

==> (- 9 1)

==> (/ 6 2)

==> (+ (* 2 4) (- 4 6))

==> (define a 3)

==> (define b (+ a 1))

==> (+ a b (* a b))
```

---

[15] A minor difference between *if* and *cond* is that, in Scheme, the *⟨e⟩* part of each *cond* clause may be a sequence of expressions, which are evaluated in sequence if the corresponding *⟨p⟩* is triggered. In an *if* combination, however, the *⟨consequent⟩* and *⟨alternative⟩* clauses must be single expressions.

```
==> (= a b)

==> (if (and (> b a) (< b (* a b)))
        b
        a)

==> (cond ((= a 4) 6)
          ((= b 4) (+ 6 7 a))
          (else 25))
```

Exercise 1-2: Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

Exercise 1-3: Show that any expression that uses *if* can be rewritten in terms of *and* and *or*. As an example, rewrite the *abs* procedure to use *and* and *or*, rather than *if* or *cond*.

Exercise 1-4: Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative order evaluation or normal order evaluation. He defines the following two procedures:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative order evaluation? What behavior will he observe with an interpreter that uses normal order evaluation? Explain your answer. (Assume that the evaluation rule for the special form *if* is the same, whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result is examined to determine whether the evaluator will continue to evaluate the consequent or the alternative expression.)

1.1.7. Example: Square Roots by Newton's Method

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

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

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

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

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

That only begs the question.

The contrast between function and procedure is a reflection of the general distinction between describing *properties* of things and describing *how to do* things, or, as it is sometimes referred to, the distinction between *declarative* knowledge and *imperative* knowledge. In mathematics we are usually concerned with declarative, or "what is" descriptions, while in computer science we are usually concerned with imperative, or "how to" descriptions.[16]

How does one compute square roots? The most common way is to use *Newton's method* of successive approximations, which says that whenever we have a guess y for the value of the square root of a number x, we can perform a simple manipulation to get a better guess -- one closer to the actual square root -- by averaging y together with x/y.[17] For example, we can compute the square root of 2 as follows. Suppose our initial guess is 1:

Guess	Quotient	Average
1	2/1 = 2	(2+1)/2 = 1.5
1.5	2/1.5 = 1.3333	(1.333 + 1.5)/2 = 1.4167
1.4167	2/1.4167 = 1.4118	(1.4167 + 1.4118)/2 = 1.4142
1.4142	...	.....

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

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

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

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

```
(define (improve guess x)
  (average guess (/ x guess)))
```

---

[16] Declarative and imperative descriptions are intimately related, as indeed are mathematics and computer science. For instance, to say that the answer produced by a program is "correct" is to make a declarative statement about the program. There is a large amount of research aimed at establishing techniques for proving that programs are correct, and much of the technical difficulty of this subject has to do precisely with negotiating the transition between imperative statements (which is how the programs are formulated) and declarative statements (which can be used to deduce things). In a related vein, an important current area in programming language design is devoted to exploring so-called "Very High Level Languages," in which one actually programs in terms of declarative statements. The idea is to make interpreters sophisticated enough so that, given "what is" knowledge specified by the programmer, the "how to" knowledge can be generated automatically. This cannot cannot be done in general, but there are important areas where progress has been made. In Chapter 4, we shall implement such a language, a "Logic Programming" language used for information retrieval.

[17] This square root algorithm is actually a special case of Newton's Method, which is a general technique for finding roots of equations. The square root algorithm itself was developed by Heron of Alexandria in the first century. We will see how to express the general Newton's Method as a Lisp procedure in section 1.3.4.

where

```
(define (average x y)
  (/ (+ x y) 2))
```

We also have to say what we mean by a guess being "good enough." The following will do for illustration, but it is not really a very good test. (See exercise 1-6.) The idea is to improve the answer until it is close enough so that its square differs from the radicand by less than a predetermined tolerance (here .001):[18]

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))
```

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

```
(define (sqrt x)
  (sqrt-iter 1 x))
```

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

```
==> (sqrt 9)
3.0001

==> (sqrt (+ 100 37))
11.7047

==> (sqrt (+ (sqrt 2) (sqrt 3)))
1.7739

==> (square (sqrt 1000))
1000.0003
```

The *sqrt* program also illustrates that the simple procedural language we have introduced so far is sufficient for writing any purely numerical program that one could write in, say, Basic or Fortran. This might seem surprising, since we haven't included in our language any *iterative* or "looping" constructs that direct the computer to do something over and over again. *Sqrt-iter*, on the other hand, demonstrates how iteration can be accomplished using no special construct other than the ordinary ability to call a procedure.[19]

Exercise 1-5: Alyssa P. Hacker doesn't see why *if* needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of *cond*?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of *if* as follows:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

---

[18] We will use the convention of naming predicates with names whose last character is a question mark. This is just a stylistic convention. As far as the interpreter is concerned, the question mark is just an ordinary symbol.

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

```
==> (new-if (= 2 3) 0 5)
5

==> (new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses *new-if* to rewrite the square root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))
```

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

Exercise 1-6: The *good-enough?* test used in computing square roots will not be very effective if we are interested in finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing *good-enough?* is to watch how *guess* changes from one iteration to the next, and to stop when the change is a very small fraction of the guess. Design a square root procedure that uses this kind of end test. Does this work better for small and large numbers?

Exercise 1-7: Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value:

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

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

1.1.8. Procedures as Black-Box Abstractions

Sqrt is our first example of a process defined by a set of mutually defined procedures. Notice that the definition of *sqrt-iter* is *recursive*; that is, the procedure is defined in terms of itself. The idea of being able to define a procedure in terms of itself may be disturbing, because it may seem unclear how such a "circular" definition could make sense at all, much less how such a definition could specify a well-defined process to be carried out by a computer. We'll address this issue more carefully in section 1.2. But first let's consider some other important points illustrated by the *sqrt* example.

Observe that the problem of computing square roots breaks up naturally into a number of subproblems: how to tell whether a guess is good enough, how to improve a guess, and so on. Each of these tasks is accomplished by a separate procedure. The entire *sqrt* program can be viewed as a cluster of procedures (shown in figure 1-2) that mirrors the decomposition of the problem into subproblems.

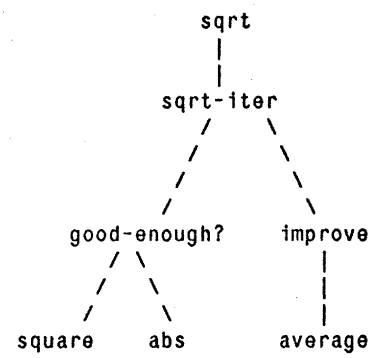


Figure 1-2: Procedural decomposition of the *sqrt* program

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

Thus, considering only the value, the following two procedures for squaring a number should be indistinguishable. Both take a number as an input and produce the square of that number as an output.[20]

```
(define (square x) (* x x))

(define (square x)
  (exp (double (log x))))

(define (double x) (+ x x))
```

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

[20] It is not even clear which of these procedures is a more efficient implementation. This depends upon the hardware available. There are machines for which the "obvious" implementation is the less efficient. Consider a machine which has extensive tables of logarithms and antilogarithms stored in a very efficient manner!

Local Names and Block Structure

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

```
(define (square x) (* x x))

(define (square y) (* y y))
```

This principle, that the meaning of a procedure should be independent of the parameter names used by its author, seems on the surface to be self-evident, but its consequences are profound. The simplest consequence is that the parameter names of a procedure must be local to the body of the procedure. For example, we used *square* in the definition of *good-enough?* in our square root routine:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))
```

The intention of the author of *good-enough?* is to determine if the square of the first argument is within a given tolerance of the second argument. We see that the author of *good-enough?* used the name *guess* to refer to the first argument and *x* to refer to the second argument. The argument of *square* is *guess*. If the author of *square* used *x* (as he did above) to refer to that argument, we see that the *x* in *good-enough?* must be a different *x* than the one in *square*. Running the procedure *square* must not modify the value of *x* which is used by *good-enough?* because that value of *x* may be needed by *good-enough?* after *square* is done computing.

If the parameters were not local to the bodies of their respective procedures, so that the *x* in *square* could be confused with the *x* in *good-enough?*, then the behavior of *good-enough?* would depend upon which version of *square* we used. Thus *square* would not be the black box we desired.

A formal parameter of a procedure has a very special role in the procedure definition, in that it doesn't matter what name the formal parameter has. Such a name is called a *bound variable* and we say that the procedure definition *binds* its formal parameters. A variable is *bound* in an expression if the meaning of the expression is unchanged by renaming the variable consistently throughout the expression to another name.[21] If a variable is not bound in an expression, we say that it is *free* in that expression. The expressions for which a binding defines a name is called the *scope* of that name. In a procedure definition, the bound variables declared as the formal parameters of the procedure have the body of the procedure as their scope.

In the definition of *good-enough?* above, *guess* and *x* are bound variables, but <, -, *abs*, and *square* are free. The meaning of *good-enough?* should be independent of the names we choose for *guess* and *x*, so long as they are distinct and different from <, -, *abs*, or *square*. (If we renamed *guess* to *abs* we would have introduced a bug by *capturing* the variable *abs*. It would have changed from free to bound.) The meaning of *good-enough?* is not independent of the names of its free variables, however. It surely depends upon the fact

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

(external to this definition) that the symbol *abs* names a procedure for computing the absolute value of a number. *Good-enough?* will compute a different function if we substitute *cos* for *abs* in its definition.

Internal definitions

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

```
(define (sqrt x)
  (sqrt-iter 1 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

(define (improve guess x)
  (average guess (/ x guess)))
```

The problem with this program is that the only procedure important to the user of *sqrt* is *sqrt*. The other procedures (*sqrt-iter*, *good-enough?*, and *improve*) only clutter up his mind. He may not define any other procedure called *good-enough?* as part of another program to work together with his square root program because he must remember that *sqrt* needs it. The problem is especially severe in the construction of large systems by many separate programmers. For example, in the construction of a large library of numerical procedures, many numerical functions are computed as successive approximations and thus would have procedures named *good-enough?* and *improve*, as auxiliary procedures. We would like to *localize* the subprocedures, hiding them inside *sqrt*, so that *sqrt* could coexist with other successive approximations, each having its own private *good-enough?* procedure. To make this possible, we allow procedures to have internal definitions that are local to that procedure. For example, in the square root problem we can write:

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) .001))

  (define (improve guess x)
    (average guess (/ x guess)))

  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))

  (sqrt-iter 1 x))
```

This works, and it is basically the right thing for solving the simplest name packaging problem. But there is a better idea lurking here. In addition to internalizing the definitions of the auxiliary procedures, we can simplify them. Since *x* is bound in the body of `sqrt` and since the definitions of `sqrt-iter`, etc. are in that scope, it is not necessary to pass *x* explicitly to each of them. Specifically, we allow *x* to be a free variable in the internal definitions. *X* then gets its value from the argument with which the enclosing procedure `sqrt` is called. This discipline is called *lexical scoping*.[22]

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) .001))

  (define (improve guess)
    (average guess (/ x guess)))

  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))

  (sqrt-iter 1))
```

From now on we will use this technique of *block structure* quite extensively to help us break up large programs into tractable pieces.[23] The block structure idea originated with the programming language Algol-60. It appears in most advanced programming languages and is an important tool for helping to organize the construction of large programs.

1.2. Procedures and the Processes they Generate

We have been introduced to the elements of programming -- to primitives, combinations, procedures, and naming. But that is not enough to say that we know how to program. Our situation is analogous to someone who has learned the rules for how the pieces move in chess, but knows nothing of typical openings, of tactics, or of strategy. Like the novice chess player, we don't yet know the *common patterns of usage* in our domain. We lack the knowledge of which moves are worth making -- which procedures are worth defining. We lack the experience to predict the consequences of making a move, or of executing a procedure.

The ability to predict, or to *pre-visualize*, the consequences of the actions under consideration is crucial in becoming an expert programmer, just as it is in any synthetic, creative activity. In becoming an expert photographer, for example, one must learn how to look at a scene and *know* how dark each region will appear on a print for each possible

---

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

[23] Be careful here. Those embedded definitions must come first in a definition. The management is not responsible for the consequences of running a program that intertwines definition and use.

choice of exposure and development conditions. Only then can one reason backwards, planning framing, lighting, exposure, and development to obtain the desired effects. So it is with programming, where we are planning the course of action to be taken by a process, and we control the process by means of a program. To become experts, we must learn to pre-visualize the processes engendered by various types of procedures. Only having developed such a skill can we learn to reliably construct programs that exhibit the desired behavior.

A procedure is a pattern for the *local evolution* of a computational process. It specifies the evolution of a process in the same way that a differential equation describes the evolution of a physical system. At each instant, the change in state of a physical system is computed from its current state according to its equations of motion. At each step, the next state of the process is computed from its current state according to the rules of interpreting procedures. Much of the theory of differential equations is concerned with describing the overall, or *global*, behavior of a system whose local evolution has been specified by a differential equation. Similarly, we would like to be able to make statements about the "overall" behavior of a process whose local evolution has been specified by a procedure. This is very difficult to do in general, but we can at least try to describe some typical patterns of process evolution.

In this section, we'll examine some common "shapes" for processes generated by simple procedures. We'll also investigate the rates at which these processes consume the important computational resources of *time* and *space*. The procedures we will be considering are very simple. Their role is like that played by photographic test patterns in photography -- as oversimplified prototypical patterns, rather than as practical examples in their own right.

1.2.1. Linear Recursion and Iteration

Let's begin by considering the factorial function, defined by

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

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

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

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

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

We can use the substitution model of section 1.1.5 to watch this procedure in action computing 6!, as shown in figure 1-3.

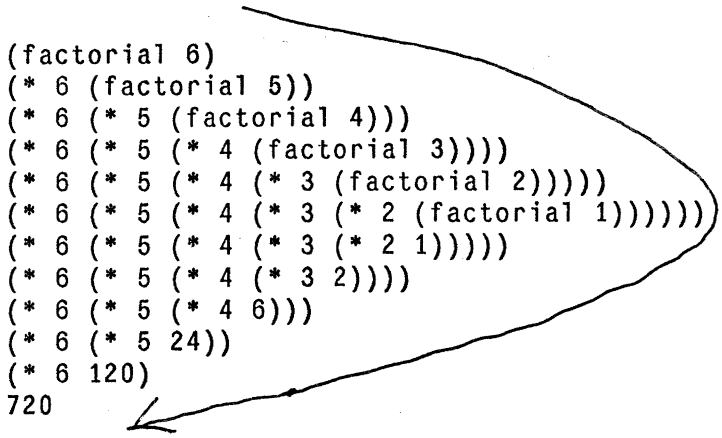


Figure 1-3: A linearly recursive process for computing 6 factorial.

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

```
product ← counter • product
counter ← counter + 1
```

together with the stipulation that the value of *factorial* is the value of the product when the counter exceeds *n*.

Once again, we can recast our description as a procedure for computing factorials:[24]

```
(define (factorial n)
  (fact-iter 1 1 n))

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

As before, we can use the substitution model to visualize the process of computing 6!, as shown in figure 1-4.

---

[24] In a real program we would probably use the block structure introduced in the last section to hide the definition of *fact-iter* and to simplify the argument passing as follows:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

We did not do this here so as to minimize the number of things to think about.

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

Figure 1-4: An iterative process for computing 6 factorial.

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

Consider the first process. The substitution model reveals a shape of expansion followed by contraction, indicated by the arrow in figure 1-3. The expansion occurs as the process builds up a chain of *deferred operations*, in this case, a chain of multiplications. The contraction occurs as the arguments to each multiplication are evaluated and the multiplication is actually performed. This type of process, characterized by a chain of deferred operations, is called a *linearly recursive process*. Notice that carrying out this process requires that the interpreter keep track of the multiplications to be performed later on. In computing *n*!, the length of the chain of deferred operations, and hence the amount of information needed to keep track of it, grows linearly with *n*.

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

Here is another way to view the contrast between the two processes. In the iterative case, the program variables provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables. Not so with the linearly recursive process. In this case there is some additional "hidden" data being maintained by the interpreter, not contained in the program variables, which keeps track of "where the process is" in negotiating the chain of deferred operations. The longer the chain, the more information to be maintained.[25]

---

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

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

One reason that the distinction between process and procedure may be confusing is that interpreters for most common languages (including Algol, Pascal, and indeed -- until recently -- most implementations of Lisp) are designed in such a way that the interpretation of *any* recursive procedure consumes an amount of memory that grows linearly with the number of procedure calls, even when the process described is, in principle, iterative. As a consequence, these languages can describe iterative processes only by resorting to the use of special-purpose "looping constructs" such as `do`, `repeat`, `until`, `for`, `while`, and so on. The interpreter we shall exhibit in Chapter 5 does not share this defect. It will execute an iterative process in constant space, even if the iterative process is described by a recursive procedure. An interpreter with this property is called *tail recursive*. With a tail recursive interpreter, iteration can be expressed using the ordinary procedure call mechanism, so that special iteration constructs are useful only as syntactic sugar.[26]

Exercise 1-8: Each of the following two procedures defines a method for adding two positive integers in terms of the more primitive operators `1+`, which increments its argument by 1, and `-1+`, which decrements its argument by 1.

```
(define (+ a b)
  (if (= a 0)
      b
      (1+ (+ (-1+ a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (-1+ a) (1+ b))))
```

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

Exercise 1-9: The following procedure computes a mathematical function called "Ackerman's function."

---

[26] Tail recursion has long been known as a compiler optimization trick. A coherent semantic basis for tail recursion was provided by Carl Hewitt [21], who explained it in terms of the "message-passing" model of computation that we shall discuss in Chapter 3. Inspired by this, Gerald Jay Sussman and Guy Lewis Steele, Jr. [39] constructed a tail-recursive interpreter for Scheme. Steele [40] later showed how tail recursion is a consequence of the natural way to compile procedure calls.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10)

(A 2 4)

(A 3 3)
```

Consider the following procedures, where *A* is the procedure defined above:

```
(define (f n) (A 0 n))

(define (g n) (A 1 n))

(define (h n) (A 2 n))

(define (k n) (* 5 n n))
```

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

1.2.2. Tree Recursion

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

0, 1, 1, 2, 3, 5, 8, 13, 21, . . .

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

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

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

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

Let us consider the pattern of this computation. In order to compute, say, *(fib 5)*, we compute *(fib 4)* and *(fib 3)*. In order to compute *(fib 4)*, we compute *(fib 3)* and *(fib 2)*. In general, the evolved process looks like a tree, as shown in figure 1-5. Notice that the branches split into 2 at each level (except at the bottom) and this reflects the fact that the *fib* procedure calls itself twice each time it is invoked.

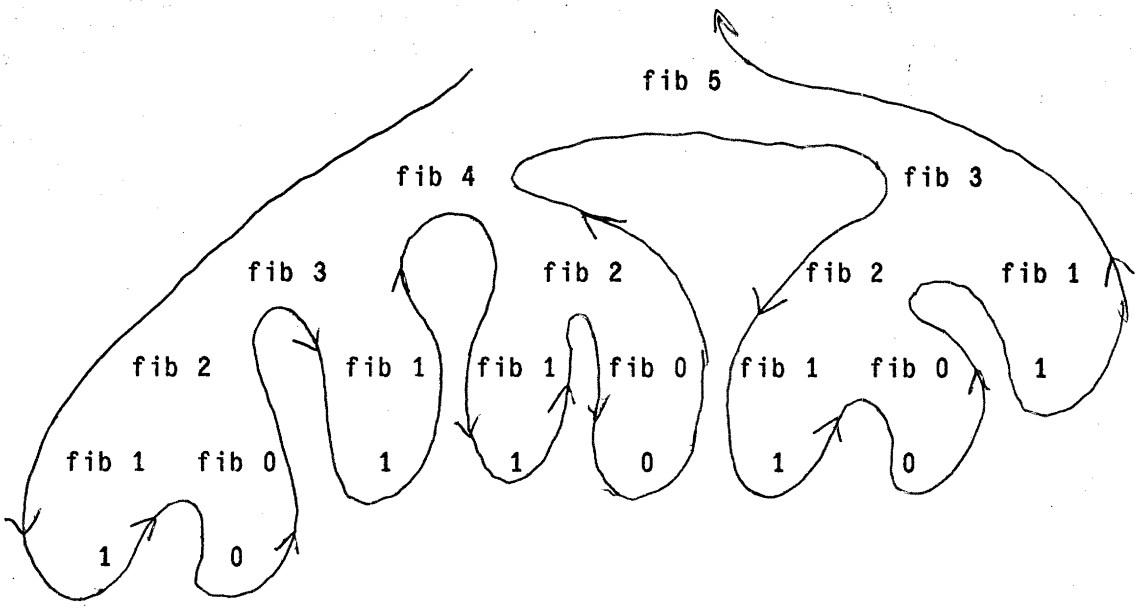


Figure 1-5: The tree-recursive process generated in computing *(fib 5)*.

This procedure is instructive as a prototypical tree-recursion, but it is a *terrible* way to compute Fibonacci numbers, because it does so much redundant computation. Notice in figure 1-5 that the entire computation of *(fib 3)* -- almost half the work -- is duplicated. In fact, it is not to hard to show that the number of times the procedure will compute *(fib 1)* or *(fib 0)* (the number of leaves in the above tree, in general) is precisely $Fib(n+1)$. And to get an idea of how bad this is, one can show that the value of $Fib(n)$ grows exponentially with n. More precisely, (see exercise 1-15) $Fib(n)$ is the closest integer to $\varphi^n/\sqrt{5}$, where

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

is the *golden ratio* that satisfies the equation

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

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

We can also formulate an iterative process for computing the Fibonacci numbers. The idea is to use a pair of integers a and b, initialized to 1 and 0, and to repeatedly apply the simultaneous transformations

```
a ← a + b
b ← a
```

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

```
(define (fib n)
  (fib-iter 1 0 n))

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

This second method for computing *fib* is a linear iteration. The difference in time required by the two methods -- one linear in *n*, one growing as fast as *Fib(n)* itself -- is enormous, even for small inputs.

One should not conclude from this that tree recursive processes are useless. For one thing, when we consider processes that operate, not on numbers, but on hierarchically structured data, we will find that tree-recursion is a natural and powerful tool.[27] But even in numerical operations, tree recursive processes can be useful in helping us to understand and design programs. Notice, for instance, that although the first *fib* procedure is much less efficient than the second one, it is more straightforward, being little more than a translation into Lisp of the definition of the Fibonacci sequence. In order to formulate the iterative algorithm we needed a bit of cleverness to notice that the computation could be recast as an iteration with three state variables.

Example: Counting change

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

This problem has a simple solution as a recursive procedure: Suppose we think of the types of coins available as arranged in some order. Then the following relation holds:

Number of ways to change amount *a* using *n* kinds of coins =
  Number of ways to change amount *a* using all but the first kind of coin
+ Number of ways to change amount *a-d* using all *n* kinds of coins
  where *d* is the denomination of the first kind of coin.

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

Thus we can recursively reduce the problem of changing a given amount to the problem of changing smaller amounts using fewer kinds of coins. You should consider this reduction

---

[27] We've already hinted at an example of this -- the interpreter itself evaluates expressions, using a tree-recursive process, as mentioned in section 1.1.3.

rule carefully, and convince yourself that we can use it to describe an algorithm if we specify the following degenerate cases:[28]

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

We can easily translate this description into a recursive procedure:

```
(define (count-change amount)
  (cc amount 5))

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

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

(The *first-denomination* procedure takes as input the number of kinds of coins available and returns the denomination of the first kind. Here we are thinking of the coins as arranged in order from smallest to largest, but any order would do as well.) Having typed in our program, we can use it to answer our original question about changing a dollar:

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

Count-change generates a tree-recursive process with redundancies similar to those in our first implementation of *fib*. (It will take quite a while for that 292 to be computed.) On the other hand, it is not so obvious how to design a better algorithm for computing the result, and we leave this problem as a challenge (exercise 1-10). The observation that a tree-recursive process may be highly inefficient but often easy to specify and to understand has led people to propose that one could get the best of both worlds by designing a "smart compiler" that can transform tree-recursive procedures into more efficient procedures that compute the

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

same result.[29]

Exercise 1-10: Design a procedure which evolves an iterative process for solving the change counting problem. For simplicity, you may wish to start by considering only 2 or 3 kinds of coins.

1.2.3. Orders of Growth

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

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

We say that $R(n)$ has *order of growth* $O(f(n))$, written $R(n) = O(f(n))$ (pronounced "Oh of $f(n)$") if there is some constant K independent of n such that

$$R(n) \leq K f(n)$$

for any sufficiently large value of n.

For instance, with the linear recursive process for computing factorial described in section 1.2.1 the number of steps grows proportionally to the input n. Thus the time required for this process grows as $O(n)$. We also saw that the space required grows as $O(n)$. For the iterative factorial the required time is still $O(n)$ but the space is $O(1)$ -- that is, constant.[30] The tree-recursive Fibonacci computation requires time $O(\varphi^n)$ and space $O(n)$.

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n steps and a process requiring $1000n$ steps are both

---

[29] This idea is not as outlandish as it may appear at first sight. One approach to coping with redundant computations is to arrange matters so that we automatically construct a table of function values as they are computed. Each time we are asked to compute the function on some input, we first look to see if the value is already stored in the table, in which case we avoid performing the redundant computation. This strategy, known as *tabulation* or *memoization*, can be implemented in a straightforward way. Tabulation can be used to transform processes that require exponential time (such as `count-change`) into processes whose space and time requirements grow linearly with the input. See exercise 3-24 of Chapter 3.

[30] These statements mask a great deal of oversimplification. For instance, when we identify process steps with "time" we are making the assumption that the amount of time needed to perform, say, a multiplication is independent of the size of the numbers to be multiplied, which is false if the numbers are sufficiently large. Similar remarks hold for the estimates of space. Just as with the design and description of a process, the analysis of a process can be carried out at various levels of abstraction.

considered to have $O(n)$ order of growth.[31] On the other hand, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the input. For an $O(n)$ process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in input size will multiply the resource utilization by a constant factor. In the remainder of section 1.2, we'll examine two algorithms whose order of growth is logarithmic, so that doubling the input size increases the resource requirement by a constant amount.

Exercise 1.11: Draw the tree illustrating the process generated by the `count-change` procedure of section 1.2.2 in making change for 11 cents. What is the order of growth of this process as the amount to be changed increases?

1.2.4. Exponentiation

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

$$b^n = b \cdot b^{n-1}$$
$$b^1 = b$$

which translates readily into the procedure

```
(define (expt b n)
  (if (= n 1)
      b
      (* b (expt b (- n 1)))))
```

This is a linear recursive process, with time and space requirements $O(n)$. Just as with factorial, we can readily formulate an equivalent linear iteration:

```
(define (expt b n)
  (exp-iter b n 1))

(define (exp-iter b counter product)
  (if (= counter 0)
      product
      (exp-iter b
                (- counter 1)
                (* b product))))
```

---

[31] Another drawback of order notation is that it provides only an *upper bound* on the growth. Because of the inequality sign in the definition, any process with order of growth $f(n)$ will also have order of growth $g(n)$ for any function g that grows faster than f. For example, any $O(n)$ process is also $O(n^2)$. So strictly speaking, we should interpret the equation $R(n) = O(f(n))$ to mean that $R(n)$ grows *at most as fast* as $f(n)$. In more careful analyses of resource utilization, one also considers estimates which say that $R(n)$ grows *at least as fast* as $f(n)$. This is expressed using the notation $R(n)$ has *order of growth* $\Omega(f(n))$, written $R(n) = \Omega(f(n))$ (pronounced "big omega of $f(n)$") which is defined to mean that there is some constant K independent of n such that

$$R(n) \geq K f(n)$$

for any sufficiently large value of n. In addition, the notation $R(n) = \theta(f(n))$ is used to mean that $R(n) = O(f(n))$ and $R(n) = \Omega(f(n))$, or, roughly, that $R(n)$ grows exactly as fast as $f(n)$.

This version requires time $O(n)$ and space $O(1)$.

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

$$b \cdot b \cdot b \cdot b \cdot b \cdot b \cdot b \cdot b$$

we can compute it using three multiplications as follows:

$$b^2 = b \cdot b$$
$$b^4 = (b^2)^2$$
$$b^8 = (b^4)^2$$

This method works fine for exponents that are powers of 2. We can also take advantage of successive squaring in computing exponentials in general if we use the rule:

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

We can express this method as a procedure:

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

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

```
(define (even? n)
  (= (remainder n 2) 0))
```

The process evolved by *fast-exp* grows logarithmically with n in both space and time. To see this, observe that computing *(fast-exp b 2n)* requires only one more multiplication than computing *(fast-exp b n)*. The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed.[32] So the number of multiplications required for an exponent of n grows about as fast as the logarithm of n to the base 2. The process has $O(\log n)$ growth.[33]

The difference between $O(\log n)$ growth and $O(n)$ growth becomes striking as n becomes large. For example, *fast-exp* for $n = 1000$ requires only 14 multiplications.[34] It is also possible to use the successive squaring idea to devise an iterative algorithm that computes exponentials in logarithmic time, although, as is often the case with iterative algorithms, this is not written down so straightforwardly as the recursive algorithm. (See exercise 1-12.)

---

[32] More precisely, the number of multiplications required is equal to one less than the log base 2 of n plus the number of 1's in the binary representation of n. This total is always less than twice the log base 2 of n. This algorithm, or more precisely, the iterative version of it (see exercise 1-12), is very ancient. It appears in the Hindu *Chandah-sutra* by Acharya Pingala, written before 200 B.C. See Knuth [24] section 4.6.3 for a full discussion and analysis of this and other methods of exponentiation.

[33] The arbitrary constant K in the definition of order notation implies that, for a logarithmic process, the base to which logarithms are taken does not matter, so all such process are described as $O(\log n)$.

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

Exercise 1-12: Design a procedure which evolves an iterative exponentiation process that uses successive squaring and works in logarithmic time, as does *fast-exp*. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant* quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

Exercise 1-13: The exponentiation algorithms in section 1.2.4 are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (assume that our language can only add, not multiply) is analogous to the *exp* procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes time linear in b. Now suppose we include, together with addition, operations *double*, which doubles an integer, and *halve*, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to *fast-exp*, which works in logarithmic time.

Exercise 1-14: Using the results of exercises 1-12 and 1-13 devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving, that works in logarithmic time.[35]

Exercise 1-15: Prove that *fib(n)* is the closest integer to $\varphi^n/\sqrt{5}$, where $\varphi = (1+\sqrt{5})/2$. (Hint: Let $\psi = (1-\sqrt{5})/2$. Use induction and the recurrence relation for the Fibonacci numbers to prove that *fib(n)* $= (\varphi^n - \psi^n)/\sqrt{5}$.) Using this fact, devise a procedure that computes Fibonacci numbers in *logarithmic* time. (Assume that there are primitive procedures *floor* and *ceiling* which return, respectively, the closest integers below and above their argument.) Explain why this method is not likely to be practical for computing *(fib n)* unless n is fairly small.

1.2.5. Greatest Common Divisors

The greatest common divisor, or GCD, of two integers a and b is defined to be the largest integer that evenly divides both a and b. For example, the GCD of 16 and 28 is 4. In Chapter 2, when we investigate how to implement rational number arithmetic, we will need to be able to compute GCDs in order to reduce rational numbers to lowest terms. (To reduce a rational number to lowest terms, we must divide both numerator and denominator by their GCD. For example, 16/28 reduces to 4/7.) One way to find the GCD of two integers is to factor them and search for common factors. But there is a famous algorithm that is much more efficient.

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

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

to successively reduce the problem of computing GCD to the problem of computing the GCD

---

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

of smaller and smaller pairs of integers. For example,

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

reduces GCD(206,40) to GCD(2,0), which is 2. It is possible to show that starting with any two positive integers and performing repeated reductions will always eventually produce a pair where the second number is 0. Then the GCD is the other number in the pair. This method for computing the GCD is known as *Euclid's Algorithm*.[36]

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

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

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

The fact that the number of steps required by Euclid's Algorithm has logarithmic growth bears an interesting relation to the Fibonacci numbers. Consider the following result:[37]

Lamé's Theorem: If Euclid's Algorithm requires k steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the kth Fibonacci number.[38]

We can use this theorem to get an order of growth estimate for Euclid's Algorithm. Let n be the smaller of the two inputs to the procedure. If the process takes k steps, then we must

---

[36] Euclid's algorithm is so called because it appears in Euclid's *Elements* (Book 7, *c.* 300 B.C.). According to Knuth [24], it may be considered to be the oldest known non-trivial algorithm. The ancient Egyptian method of multiplication (exercise 1-14) is surely older, but, as Knuth explains, Euclid's algorithm is the oldest known to have been presented as a general algorithm, rather than only as a set of illustrative examples.

[37] This theorem was proved in 1845 by Gabriel Lamé, a French mathematician and engineer known chiefly for his contributions to mathematical physics.

[38] To prove the theorem, we consider pairs (a_k, b_k) for which Euclid's algorithm terminates in k steps. The proof is based on the claim that if $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ are three successive pairs in the reduction process, then we must have $b_{k+1} \geq b_k + b_{k-1}$

To verify the claim, consider that a reduction step is defined by applying the transformation:

$a_{k-1} = b_k$
b_{k-1} = remainder of a_k divided by b_k

The second equation means that $a_k = qb_k + b_{k-1}$ for some positive integer q. And since q must be at least 1 we have $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$ But in the previous reduction step, we have $b_{k+1} = a_k$. Therefore, $b_{k+1} = a_k \geq b_k + b_{k-1}$ This verifies the claim.

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

have $n \geq \text{Fib}(k) \approx \varphi^k$. Therefore the number of steps k must be less than the logarithm (to the base φ) of n. Hence the order of growth is $O(\log n)$.

Exercise 1-16: The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative *gcd* procedure given in section 1.2.5, which has logarithmic growth. Suppose we were to interpret this procedure using normal order evaluation, as discussed in section 1.1.5. Using the substitution method (for normal order), illustrate the process generated in evaluating *(gcd 206 40)*. In general, what is the order of growth in time resources for *gcd* using normal order evaluation? (Assume that the time required is proportional to the number of *remainder* operations performed.)

1.2.6. Example: Testing for Primality

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

Searching for divisors

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

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond
   ((> (square test-divisor) n) n)
   ((divides? test-divisor n) test-divisor)
   (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

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

```
(define (prime? n)
  (= n (smallest-divisor n)))
```

The end test for *find-divisor* is based on the fact that if n is not prime, it must have a divisor less than or equal to $\sqrt{n}$.[39] This means that the algorithm need only test divisors between 1 and $\sqrt{n}$. Consequently the number of steps required to identify n as prime will have order of growth $O(\sqrt{n})$.

---

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

The Fermat test

The *O(log n)* primality test is based on a result from number theory known as *Fermat's Little Theorem*:

Fermat's Little Theorem: If n is a prime number and a is any positive integer less than n, then a raised to the n-th power is congruent to a modulo n.

(Two numbers are said to be *congruent modulo n* if they both have the same remainder when divided by n.)

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

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

```
(define (expmod b e m)
  (cond ((= e 1) b)
        ((even? e)
         (remainder (square (expmod b (/ e 2) m))
                    m))
        (else
         (remainder (* b (expmod b (- e 1) m))
                    m))))
```

This is very similar to the *fast-exp* procedure of section 1.2.4. Observe that it uses successive squaring, so that the number of steps grows logarithmically with the exponent.[40]

The Fermat test is performed by choosing at random a number a between 2 and $n-1$ inclusive and checking whether the remainder modulo n of n-th power of a is equal to a. The random number a is chosen using the procedure *random*, which is included as a primitive in Scheme. *Random* returns a non-negative integer less than its input. Hence, to obtain a random number between 2 and $n-1$, we call *random* with an input of $n-2$ and add 2 to the result.

```
(define (fermat-test n)
  (define a (+ 2 (random (- n 2))))
  (= (expmod a n n) a))
```

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

---

[40] The reduction steps in the cases where the exponent e is greater than 1 are based on the fact that, for any integers x, y, and m, we can find the remainder of x times y modulo m by computing separately the remainders of x modulo m and y modulo m, multiplying these, and then taking the remainder of the result modulo m. For instance, in the case where e is even, we compute the remainder of $b^{e/2}$ modulo m, square this, and take the remainder modulo m. This technique is useful because it means we can perform our computation without ever having to deal with numbers much larger than m. (Compare exercise 1-21.)

```
(define (fast-prime? n times)
  (or (= times 0)
      (and (fermat-test n)
           (fast-prime n (- times 1)))))
```

(This procedure uses *and* and *or* in a tricky way. Remember that these operators are special forms. The second clause of the *and* will be evaluated only if the first one is true, and the second clause of the *or* will be evaluated only if the first one is false. This implies that *fast-prime?* will return immediately if it ever finds *(fermat-test n)* to be false.)

Probabilistic methods

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

Unfortunately, this assertion is not quite correct, because there do exist numbers that fool the Fermat test: numbers n that are not prime and yet have the property that a^n is congruent to a modulo n for all integers $a < n$. Such numbers are extremely rare, however, so the Fermat test is quite reliable in practice. Nevertheless, the possibility of error still exists, and because of this, mathematicians until recently tended to regard the Fermat test as a good way to show that a number is *not* prime, but not an adequate method for showing that a number *is* prime.[41]

Over the past few years, mathematicians have discovered variations of the Fermat test that cannot be fooled. In these tests, as with the Fermat method, one tests the primality of an integer n by choosing a random integer $a < n$ and checking the value of some quantity $F(a,n)$ that can be computed in logarithmic time. (See exercise 1-23 for an example of such a test.) On the other hand, unlike with the Fermat test, one can *prove* that for any n, $F(a,n)$ will not have the right value for most of the integers $a < n$ unless n is prime. This means that if n passes the test for some random choice of a, we *know* that the chances are better than even that n is prime. If n passes the test for 2 random choices of a then the odds are better than 4 to 1 that n is prime. And by running the test with more and more randomly chosen values of a we can make the probability of error as small as we like.

The difference between these methods and the Fermat test is not significant for practical purposes.[42] On the other hand, the existence of tests for which one can *prove* that the chance of error becomes arbitrarily small sparked interest in algorithms of this type, which

---

[41] Numbers that fool the Fermat test are called *Carmichael numbers*, and little is known about them, other than that they are extremely rare. There are 16 Carmichael numbers below 100,000. The smallest few are 561, 1105, 1729, 2465, 2821, 6601.

[42] In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

have come to be known as *probabilistic algorithms*. There is currently a great deal of research activity in this area, and probabilistic algorithms have been fruitfully applied to many fields.[43]

Exercise 1-17: Implement the `smallest-divisor` procedure of section 1.2.6 and use it to find the smallest divisor of each of the following numbers: 199; 1999; 19999.

Exercise 1-18: Most Lisp implementations include a primitive called `runtime` which returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following `timed-prime-test` procedure, when called with an integer *n*, prints *n* and checks to see if *n* is prime. If *n* is prime, the procedure prints three stars, followed by the number of microseconds used in performing the test.

```
(define (timed-prime-test n)
  (define start-time (runtime))
  (define found-prime? (prime? n))
  (define elapsed-time (- (runtime) start-time))
  (print n)
  (cond (found-prime? (print " *** ")
                      (print elapsed-time))
        (else nil)))
```

Using this procedure, write a procedure `search-for-primes` which checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth $O(\sqrt{n})$ you should expect that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Does your timing data bear this out? How well does the data for 100,000 and 1,000,000 support the $\sqrt{n}$ prediction?

Exercise 1-19: The `smallest-divisor` procedure described in section 1.2.6 is doing lots of needless testing. For after it checks to see if the number is divisible by 2, there is no point checking to see if it is divisible by any larger even numbers. This suggests that the values used for `test-divisor` should not be 2, 3, 4, 5, 6, 7, ..., but rather 2, 3, 5, 7, 9, ... To implement this change, define a procedure `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest-divisor` procedure to use `(next test-divisor)` instead of `(+ test-divisor 1)`. With `timed-prime-test` incorporating this modified version of `smallest-divisor`, run the test for each of the 12 primes found in exercise 1-18. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

Exercise 1-20: Implement the Fermat test as described in section 1.2.6. Modify the `timed-prime-test` procedure of exercise 1-18 to use the Fermat method, and test each of the 12 primes you found in that exercise. Since the Fermat test has $O(\log n)$ growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Does your data bear this out? Can you explain any discrepancy you find?

Exercise 1-21: Alyssa P. Hacker complains that we went to a lot of extra work in writing *expmod*. After all, she says, since we already know how to compute exponentials, we could have simply written

---

[43] One of the most striking applications of probabilistic prime testing has been to the field of cryptography. While it is currently computationally infeasible to factor an arbitrary 200-digit number, the primality of such a number can be checked in a few seconds with the Fermat test. This fact forms the basis of a technique for constructing "unbreakable codes" suggested in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adelman [36]. Because of this and related developments, the study of prime numbers, once considered to be the epitome of a topic in "pure" mathematics to be studied only for its own sake, now turns out to have important practical applications to cryptography, electronic funds transfer, and information retrieval.

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

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

Exercise 1-22: Louis Reasoner is having great difficulty doing exercise 1-20. His *fast-prime?* test seems to run more slowly than his *prime?* test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the *expmod* procedure to use an explicit multiplication, rather than calling *square*:

```
(define (expmod b e m)
  (cond ((= e 1) b)
        ((even? e)
         (remainder (* (expmod b (/ e 2) m)
                       (expmod b (/ e 2) m))
                    m))
        (else
         (remainder (* b (expmod b (- e 1) m))
                    m))))
```

"I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the *O(log n)* process into an *O(n)* process." Explain.

Exercise 1-23: One of the variants of the Fermat test which cannot be fooled was discovered in 1977 by Robert Solovay and Volker Strassen [38]. It proceeds by choosing a random number $a < n$, checking that $GCD(a,n) = 1$ and then computing a number-theoretic quantity called the *Jacobi symbol J(a,n)* which is equal to ± 1. If n is prime then $J(a,n)$ is always congruent modulo n to $a^{(n-1)/2}$ for any a such that $GCD(a,n) = 1$. If n is not prime, then it can be proved that this relation does not hold for at least half the numbers $a<n$. Thus if we find that the relation does not hold for some randomly chosen a, we can assert that the chances are better than even that n is not prime. The Jacobi symbol can be computed by using the reductions:

$$J(a,n) = \begin{cases} 1 & \text{if } a=1 \\ J(a/2,n)*(-1)^{(n^2-1)/8} & \text{if } a \text{ is even} \\ J(\text{remainder}(n,a),a)*(-1)^{(a-1)(n-1)/4} & \text{otherwise} \end{cases}$$

Implement the Solovay-Strassen test as a procedure that runs in *O(log n)* time.

1.3. Formulating Abstractions with Higher Order Procedures

We've seen that procedures are, in effect, abstractions that describe compound operations on numbers independently of the particular numbers. For example, when we define

```
(define (cube x) (* x x x))
```

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

```
(* 3 3 3)
(* x x x)
(* y y y)
   .
   .
   .
```

and never mentioning *cube* explicitly. This would place us at a serious disadvantage, forcing us to work always at the level of the particular operations that happen to be primitives in the

language (multiplication, in this case), rather than in terms of higher level operations. Our programs would be able to compute cubes, but our language would lack the ability to express the *concept* of cubing. One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning *names* to common patterns, and then to work in terms of the abstractions directly. Procedures provide this ability. This is why all but the most primitive programming languages include mechanisms for defining procedures.

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

1.3.1. Procedures as Parameters

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

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

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

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

The third computes the sum of a sequence of terms in the following series, which converges to $\pi/8$ (very slowly):[44]

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

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

These three procedures clearly share a common underlying pattern. They are for the most part identical, differing only in the name of the procedure, the function of *a* used to compute the term, and the function that provides the next value of *a*. We could generate each of the

---

[44] This formula, usually written in the equivalent form

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \ldots$$

is due to Leibnitz.

procedures by filling in slots in the same template:

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b))))
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Indeed, mathematicians have long ago identified the abstraction of *summation of a series* and have invented "sigma notation"

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

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

Similarly, as program designers, we would like our language to be powerful enough so that we can write a procedure that expresses the *concept of summation itself* rather than only writing procedures that compute particular sums. And we can readily do so in our procedural language by taking the common template shown above and transforming the "slots" into formal parameters:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

Notice that *sum* takes as its arguments upper and lower bounds *a* and *b* together with *procedures* `term` and `next`. We can use *sum* just as we would any procedure. For example, we can use it to define `sum-cubes`:

```
(define (sum-cubes a b)
  (sum cube a 1+ b))
```

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

```
==> (sum-cubes 1 10)
3025
```

We could also define `pi-sum` in the same way:

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

Using these procedures, we could get an approximation to π:

```
==>(* 8 (pi-sum 1 1000))
3.13592
```

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

$$\int_a^b f = [f(a+dx/2) + f(a+dx+dx/2) + f(a+2dx+dx/2) + \ldots] dx$$

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

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2)) add-dx b)
     dx))

==> (integral cube 0 1 .001)
0.250000063
```

Exercise 1-24: The *sum* procedure above generates a linear recursion. If we like, we can rewrite the procedure so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if <??>
        <??>
        (iter <??>
              <??>)))
  (iter <??> <??>))
```

Exercise 1-25: [45] The *sum* procedure of section 1.3.1 is only the simplest of a vast number of similar abstractions that can be captured as higher order procedures. Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Write the procedure in two forms, one which generates a recursive process and one which generates an iterative process. Show how to define `factorial` in terms of `product`. Also use `product` to compute approximations to π using the formula:[46]

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

Exercise 1-26: Show that *sum* (section 1.3.1) and `product` (exercise 1-25) are both special cases of a still more general notion called `accumulate` which combines a collection of terms, using some general

---

[45] The intent of exercises 1-25 through 1-27 is to demonstrate the expressive power that is attained by using an appropriate abstraction to consolidate many seemingly disparate operations. However, while accumulation and filtering are elegant ideas, our hands are somewhat tied in using them at this point, since we do not yet have data structures to provide suitable *means of combination* for these abstractions. We will return to these ideas in Chapter 3 when we study data structures called *streams*. Streams are interfaces that allow us to combine filters and accumulators to build even more powerful abstractions. We will see in section 3.4.2 how these methods really come into their own as a powerful and elegant approach to designing programs.

[46] This formula was discovered by the English mathematician John Wallace, who lived from 1616 to 1703.

accumulation function:

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

Accumulate takes as parameters the same term and range specifications as *sum* and *product*, together with a *combiner* procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a *null-value* that specifies what initial value to use when the terms run out. Write *accumulate* (both in recursive and iterative forms) and show how *sum* and *product* can both be defined as simple calls to *accumulate*.

Exercise 1-27: You can obtain an even more general version of accumulate by introducing the notion of a *filter* on the terms to be combined. That is, do not combine all the terms in the range, but only those that satisfy a specified condition. The resulting *filtered-accumulate* abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write *filtered-accumulate* as a procedure. Show how to express, using *filtered-accumulate*:

a. the sum of the squares of the prime numbers in given interval a to b (Assume you have a *prime?* predicate already written.)

b. the product of all the positive integers $a < n$ such that GCD$(a,n) = 1$.

1.3.2. Constructing Procedures using LAMBDA

In using *sum* in section 1.3.1, it seems terribly awkward to have to define trivial procedures such as *pi-term* and *pi-next*, just so we can use them as inputs to our higher order procedure. Rather than defining names *pi-next* and *pi-term* (even if in a local environment), it would be more convenient to have a way to directly specify "the procedure that returns its input incremented by 4," and "the procedure that returns the reciprocal of its input times its input plus 2." We can do this by introducing the special form *lambda*, which can be thought of as a "*define* anonymous." Using *lambda* we can describe what we want as

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

and

```
(lambda (x) (/ 1 (* x (+ x 2))))
```

Then our *pi-sum* procedure can be expressed without defining any auxiliary procedures as

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
```

Again, using *lambda*, we can write the *integral* procedure without having to define the auxiliary procedure *add-dx*. In addition, we can include the increment *dx* as a parameter to *integral*:

```
(define (integral f a b dx)
  (* (sum f
          (+ a (/ dx 2))
          (lambda (a) (+ a dx))
          b)
     dx))
```

```
==> (integral cube 0 1 .01)
0.249987492

==> (integral cube 0 1 .001)
0.250000063
```

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

In general, *lambda* is used to define procedures in the same way as *define*, except that no name is specified for the procedure being defined.

(lambda *<formal-parameters>* *<body>*)

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

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

is equivalent to

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

As with any expression which has as its value a Lisp procedure, a *lambda* form can be used as the operator in a combination, such as:

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

or, more generally, in any context where we would normally use a procedure name.[47]

Using LET to define local variables

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

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

which we could also express as

$$a = 1+xy$$

$$b = 1-y$$

$$f(x,y) = xa^2 + yb + ab$$

In writing a procedure to compute f, we would like to include as local variables not only x and y, but also the names of intermediate quantities like a and b. One natural way to accomplish

---

[47] It would be clearer and less intimidating to people learning Lisp if one used a name more obvious than *lambda*, such as *procedure*. But the convention is firmly entrenched. The notation is adopted from the λ-*calculus* (lambda-calculus), a mathematical formalism introduced in 1941 by the mathematical logician Alonzo Church [6]. Church developed the λ-calculus to provide a rigorous foundation for studying the notions of function and function application. As such, the λ-calculus has become a basic tool for mathematical investigations of the semantics of programming languages.

this is to *define* these expressions locally:

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

An alternative is to use an auxiliary procedure to *bind* the local variables, instead of *defining* them:

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

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

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

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

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

The general form of *let* is

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      .
      .
      .
      (<varn> <expn>))
  <body>)
```

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

syntax for:

```
((lambda (<var_1> . . . <var_n>)
   <body>)
 <exp_1>
 .
 .
 .
 <exp_n>)
```

Notice that no new mechanism is required in the interpreter in order to provide local variables. `Let` is simply syntactic sugar for the underlying `lambda`.

A `let` construct (or the equivalent `lambda` expression) is often preferable to `define` for making local variables for several reasons.

- `Let` allows one to construct expressions that bind variables as locally as possible to where they are to be used. For example, one can use a `let` expression as an operand of a combination. The other operands and the operator will be evaluated in the environment outside of the `let`, but the local variables bound by the `let` will be available to help compute the value of that operand of the combination. For example, we could write

```
(+ (let ((x (1+ y)))
     (+ x (* x y)))
   y)
```

- In a `let` expression, the variables are bound *simultaneously*, using values computed *outside* the scope of the `let`, rather than being bound in sequence. This makes a difference when the expressions that provide the values for the `let` local variables depend upon variables having the same names the `let` variables themselves. For example, in an environment where *x* is bound to 2 the expression

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

will have the value 12, because, inside the scope of the `let`, *x* will be bound to 3 and *y* will be bound to 4 (which is the *original* *x* plus 2). In contrast evaluating the sequence

```
(define x 3)
(define (y (+ x 2))

(* x y)
```

will result in 15 as the value of the last expression, since *x* will be bound to 3 and *y* will then be bound to 5.

Exercise 1-28: Suppose we define the following procedure

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

Then we have

```
==> (f square)
4
```

```
==> (f (lambda (z) (* z (+ z 1))))
6
```

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

Exercise 1-29: Ben Bitdiddle writes a program to approximate π using the equation:

$$\frac{\pi}{4} = \arctan 1 = \int_0^1 \frac{dx}{1 + x^2}$$

Ben defines the procedure:

```
(define (pi dx)
  (* 4
     (integral (lambda (x) (/ 1 (+ 1 (square x))))
               0
               1
               dx)))
```

and proceeds to produce more and more accurate approximations to π by using smaller and smaller values of *dx*:

```
==> (pi .1)
3.14242598

==> (pi .01)
3.14160103

==> (pi .001)
3.14159274
```

Ben's classmate Eva Lu Ator decides she'd like to try this herself, so she sits down at the next terminal, types in the same *pi* procedure and runs it. To her amazement, the results are slightly different:

```
==> (pi .1)
3.14242595

==> (pi .01)
3.141601

==> (pi .001)
3.14159256
```

In trying to figure out what is going on, Ben and Eva make a careful comparison of all the procedures in their programs. The only difference they find is that Ben has defined *integral* to use the recursive version of *sum* from section 1.3.1 while Eva has used the iterative version. Can this account for the difference in their results? If so, how? (Hint/Warning: This is a "trick question," that depends on properties of computer arithmetic that we have not mentioned explicitly.)

1.3.3. Procedures as General Methods

We introduced compound procedures in section 1.1.4 as a mechanism for abstracting useful numerical operations so as to make them independent of the particular numbers involved. With higher order procedures, such as the *integral* procedure of section 1.3.1, we began to see a more powerful kind of abstraction -- procedures used to express general methods of computation, independently of the particular mathematical functions involved. In this section we discuss two more elaborate examples -- general methods for finding zeroes and maxima of functions -- and show how these methods can be expressed directly as procedures.

Finding roots of equations by the half-interval method

The half-interval method is a simple but powerful technique for finding roots of an equation $f(x) = 0$, where f is a continuous function. The idea is that if we are given points a and b such that $f(a) < 0 < f(b)$ then f must have at least one zero between a and b. To locate a zero we let x be the average of a and b and compute $f(x)$. If $f(x) > 0$ then f must have a zero between a and x. If $f(x) < 0$ then f must have a zero between x and b. Continuing in this way, we can identify smaller and smaller intervals on which f must have a zero. When we reach a point where the interval is small enough, the process stops. Since the interval of uncertainty is reduced by half at each step of the process, the number of steps required grows as $O(\log L/T)$, where L is the length of the original interval and T is the error tolerance, that is, the size of the interval we will consider "small enough."

Here is a procedure that implements this strategy:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

We assume that we are initially given the function f together with points on which its values are negative and positive. We first compute the midpoint of the two given points. Next we check to see if the given interval is small enough, and if so, we simply return the midpoint as our answer. Otherwise, we compute as a test value the value of f at the midpoint. If the test value is positive, then we continue the process with a new interval running from the original negative point to the midpoint. If the test value is negative, we continue with the interval from the midpoint to the positive point. Finally, there is the possibility that the test value is 0, in which case the midpoint is itself the root we are searching for.

To test whether the endpoints are "close enough" we can use a procedure similar to the one used in section 1.1.7 for computing square roots.

```
(define (close-enough? x y)
  (< (abs (- x y)) .001))
```

Finally, we can use the *search* procedure in a procedure that takes as inputs the function, together with two endpoints. This checks to see which of the endpoints has a negative function value and which has a positive value, and calls the *search* procedure accordingly. If the function has the same sign on the two given points, the half-interval method cannot be used, in which case the procedure signals an error:[48]

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

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

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

```
==> (half-interval-method sin 2 4)
3.14111328
```

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

```
==> (half-interval-method (lambda (x)
                            (- (* x x x) (* 2 x) 3))
                          1
                          2)
1.8931
```

Finding the maximum of a unimodal function

Suppose we are given a function *f* defined on some interval and we wish to find, to within a tolerance *T*, the point on the interval at which *f* attains its maximum value. One straightforward way to do this is to evaluate the function at points along the interval that are evenly spaced a distance *T* apart and to pick the one that has the maximum value. This *exhaustive search* procedure requires evaluating the function at $O(L/T)$ points, where *L* is the length of the interval. This is surely not a very effective method if *T* is small. Fortunately, for many classes of functions, there are much better techniques for locating the maximum. The method we shall discuss here applies to functions that are *unimodal*; that is, functions that are known to have only one "bump" on the interval in question. More formally, a unimodal function *f* with a maximum on an interval from *a* to *b* has the property that there is some point *m* on the interval such that *f* is increasing between *a* and *m* and decreasing between *m* and *b*.

For unimodal functions, there is a process that will find the maximum, using $O(\log(L/T))$ function evaluations. The idea is to evaluate *f* at two intermediate points *x* and *y* on the interval (with $x<y$). Then, if $f(x)$ is greater than $f(y)$, we can assert that the maximum must lie on the interval between *a* and *y*, while if $f(x)$ is less than $f(y)$ we can assert that the maximum lies between *x* and *b*. So if we choose *x* and *y* to lie towards the middle of the interval, we can with two function evaluations cut the interval of uncertainty roughly in half, as shown in figure 1-6. Repeatedly performing this cutting will produce an interval of size smaller than *T* in $O(\log(L/T))$ steps.

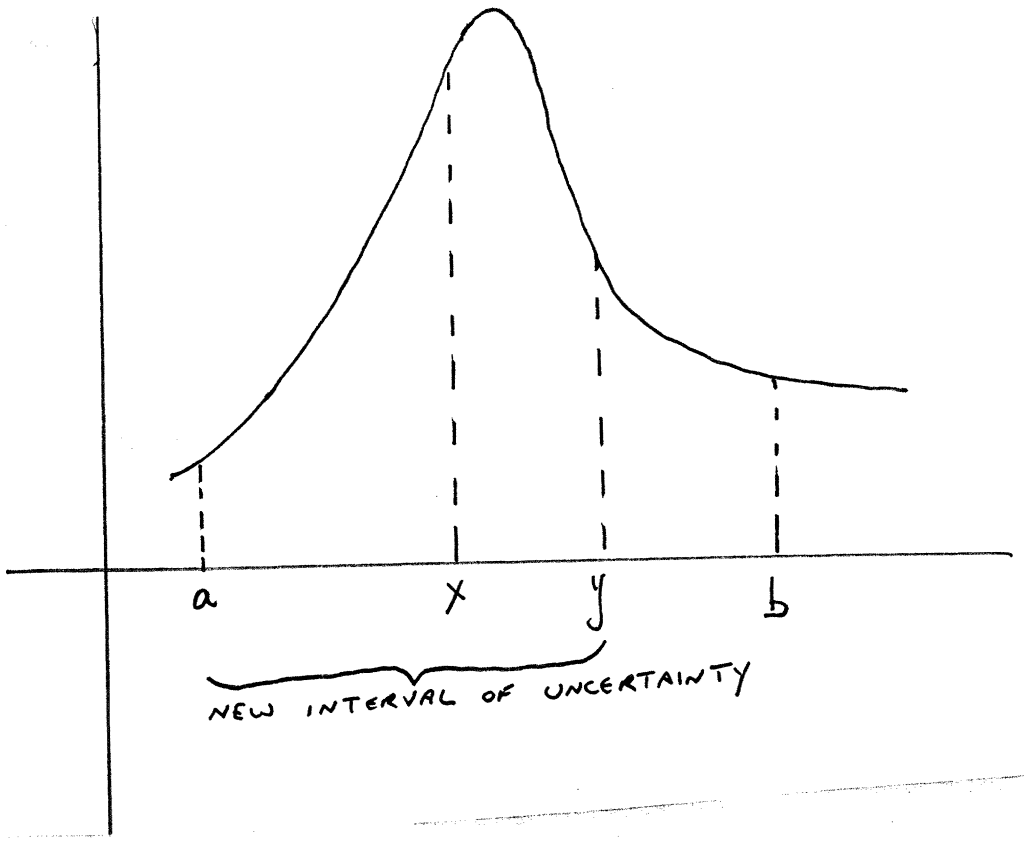


Figure 1-6: Searching for the maximum of a unimodal function

But we can be cleverer yet. Notice that we need to compare the function value at two intermediate points x and y in order to reduce the interval. Thus each reduction step requires that we find the value of the function at two intermediate points. But suppose we could arrange things so that we could guarantee that one of the intermediate values had been already generated by the previous reduction step. Then each new reduction step would require only *one* new function evaluation. If the function is difficult to compute this would represent a substantial savings, making our search twice as efficient.

One way to accomplish this, known as the *golden section method*, is to choose the intermediate points x and y to lie at certain fixed fractions along the interval. That is, we always choose x to lie at a fraction p of the way from a to b, and y to lie at a fraction q of the way from a to b:

$$x - a = p(b - a)$$
$$y - a = q(b - a)$$

The reason this is called the golden section method is that we choose the fraction q to be

$$q = (\sqrt{5} - 1)/2 \approx 0.618$$

which is precisely the reciprocal of the golden ratio, which described the order of growth of the Fibonacci numbers. The number p is chosen to be q^2. If we always use this rule to choose the intermediate values x and y, then we can guarantee that one of the intermediate values to

be chosen for the next round will be either the x or the y for the current round.[49]

In summary, we carry out the golden section method by choosing as our two intermediate points a point y which lies a fraction of the way from a to b equal to one over the golden ratio, and a point x which lies at a fraction of one over the golden ratio squared. If $f(x) > f(y)$ then the new interval of uncertainty is from a to y, x will serve as the "y point" for this interval, and we should compute a new "x point." If $f(x) < f(y)$ then the new interval of uncertainty is from x to b, y will serve as the "x point" for this interval and we should compute a new "y point." In either case, the interval of uncertainty is reduced to at most .618 of its previous length. We repeat the process, reducing the interval over and over, until the endpoints are close enough for our purposes, in which case we can return any intermediate point on the interval (say, x).

The golden section method can be implemented as an iterative process described by the following procedure *reduce* which maintains state variables for a, x, y, b, and the values of f at x and y.

```
(define (reduce f a x y b fx fy)
  (cond ((close-enough? a b) x)
        ((> fx fy)
         (let ((new (x-point a y)))
           (reduce f a new x y (f new) fx)))
        (else
         (let ((new (y-point x b)))
           (reduce f x y new b fy (f new))))))
```

Notice (and this is the whole point of the method) that we have to compute the value of f at only one new point on each iteration. The new points are computed at the appropriate ratio along the interval. The procedures to do this take as arguments the endpoints of the new interval:

---

[49] Here is a proof of this fact: Suppose that $f(x) > f(y)$. Then our new interval of uncertainty will run from a to y. If we want to use our old x as the intermediate value that lies a fraction q of the way from a to y then we should have $x - a = q(y - a)$ or

$$p(b - a) = q(y - a) = q^2(b - a)$$

which implies that $p = q^2$. Now consider the other case, where $f(x) < f(y)$. Then our new interval will run from x to b. If we want to use our old y as the intermediate point at a fraction p of the distance along this interval, we should have $y - x = p(b - x)$, or

$$(y - a) - (x - a) = p[(b - a) - (x - a)]$$

or, substituting for $x-a$ and $y-a$,

$$q(b - a) - p(b - a) = p[(b - a) - p(b - a)]$$

This reduces to

$$q - p = p - p^2$$

Combining this with the relation $p = q^2$ that we derived above yields

$$q - q^2 = q^2 - q^4$$

which simplifies to

$$1 = q^2 + q$$

The number q that satisfies this relation is the reciprocal of the golden ratio.

```
(define (x-point a b)
  (+ a (* golden-ratio-squared (- b a))))

(define (y-point a b)
  (+ a (* golden-ratio (- b a))))
```

where the golden-ratio constants are computed by

```
(define golden-ratio
  (/ (- (sqrt 5) 1) 2))

(define golden-ratio-squared (square golden-ratio))
```

Finally, we initialize the process with a procedure that takes as inputs the function we wish to maximize together with the endpoints of the interval in question and calls *reduce* after setting up the initial x and y points of the interval:

```
(define (golden f a b)
  (let ((x (x-point a b))
        (y (y-point a b)))
    (reduce f a x y b (f x) (f y))))
```

As a test, we can use our procedure to approximate π as twice the maximum point $\pi/2$ of the sine function on the interval 0 to 3:

```
==> (* 2 (golden sin 0 3))
3.14143
```

Exercise 1-30: When we introduced the golden section method for finding the maximum of a function on an interval, we also mentioned the brute force method of evaluating the function at evenly-spaced points along the interval and choosing the largest value. Assuming you have a procedure *max* that returns the larger of its two inputs, show how a brute-force search for the maximum value can be implemented as a single call to *accumulate* (exercise 1-26).

Exercise 1-31: In finding the maximum of a unimodal function, how much faster is the golden section method than the brute force method of evaluating the function at equally-spaced points along the interval and choosing the largest? In particular, suppose we want to find the maximum point of a function on the interval from 0 to 1 with an accuracy to within .001. How many function evaluations would be required using brute force? How many using the golden section method?

Exercise 1-32: A number *x* is called a *fixed point* of a function *f* if *x* satisfies the equation *f(x) = x*. For some functions *f* (the cosine function is an example) we can locate a fixed point by beginning with an initial guess and applying *f* repeatedly:

f(x), f(f(x)), f(f(f(x))), ...

until the value does not change very much. Using this idea, design a procedure *fixed-point* that takes as inputs a function and an initial guess and produces an approximation to a fixed point of the function. Test your procedure by evaluating the expression

```
(fixed-point cos 1)
```

to produce a fixed point of the cosine function.

1.3.4. Procedures as Returned Values

The previous examples show how the ability to pass procedures as parameters significantly enhances the expressive power of our programming language. We obtain even more expressive power if we have the ability to create procedures whose returned values are

themselves procedures.

Consider the statement that "the derivative of x^3 is $3x^2$." This says that the derivative of the *function* whose value at x is x^3 is another *function*, namely the function whose value at x is $3x^2$. In particular, "derivative" itself can be regarded as an operator which, when applied to a function f, returns another function Df. To describe "derivative" we can say that, if f is a function and dx is some number, then the derivative Df of f is the function whose value at any number a is given (in the limit of small dx) by

$$Df(x) = \frac{f(x+dx) - f(x)}{dx}$$

Using *lambda*, we can express the derivative formula as the procedure

```
(lambda (x)
  (/ (- (f (+ x dx)) (f x))
     dx))
```

where *dx* is some small number.

Going further, we can express the idea of derivative *itself* as the procedure

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

This is a procedure that takes as its argument a procedure *f* and returns as its value a procedure (produced by the *lambda*) which, when applied to a number *a* will produce an approximation to the derivative of *f* at *a*.

We can use our *deriv* procedure as follows to approximate the derivative of the *cube* function at 5 (whose exact value is 75):

```
==> ((deriv cube .001) 5)
75.15
```

Observe that the operator of the second combination above is itself a combination, because the procedure to be applied to 5 is the result of *deriv* applied to *cube*.

Newton's method for arbitrary functions

We can use *deriv* to build a procedure that implements *Newton's method* for finding the roots of a differentiable function. This says that if y is an approximation to a root of the function f, then a better approximation is given by:

$$y - \frac{f(y)}{Df(y)}$$

This generalizes the formula that we used in section 1.1.7 for computing square roots.[50] Now, however, we are trying to compute, not only square roots, but roots of any function. It is a general method, like the half-interval method that we described in section 1.3.3.

We can implement Newton's method as a straightforward generalization of the square root program of section 1.1.7. As before, we start with an initial guess and improve it until it is good enough:

```
(define (newton f guess)
  (if (good-enough? guess f)
      guess
      (newton f (improve guess f))))
```

Improving the guess is done using the formula given above:

```
(define (improve guess f)
  (- guess (/ (f guess)
              ((deriv f .001) guess))))
```

Finally, a guess is good enough when the value of the function at that point is very small:

```
(define (good-enough? guess f)
  (< (abs (f guess)) .001))
```

Having defined these procedures, we can now try Newton's method with any function. For example, we can approximate the value of x for which x is equal to cos x, starting with an initial guess of 1:

```
==> (newton (lambda (x) (- x (cos x))) 1)
0.7391
```

The idea of procedures as returned values may take some getting used to, or seem little more than a mathematical trick. But the increase in expressive power in a language that can return procedure values is enormous, because this means that we can compute not only with *particular* procedures, but with procedures that can evolve in response to the ongoing computation. This ability lies at the root of some powerful programming techniques which we shall meet in later chapters.[51]

> **Exercise 1-33:** If f is a numerical function and n is a positive integer, then we can form the nth *repeated application* of f, that is, the function whose value at x is $f(f(\ldots(f(x))\ldots))$. For example, if $f(x) = x + 1$, then the nth repeated application of f produces the function g where $g(x) = x + n$. If f is the operation of squaring a number, then the nth repeated application produces the operation that

---

[50] Newton's method does not always converge to an answer, but it can be shown that, in favorable cases, each iteration of the Newton formula doubles the number of digits accuracy of the approximation to the root. In such cases, Newton's method will converge much more rapidly than the half-interval method. In the case of square roots (which is a favorable case for Newton's method), we are trying to find a zero of the function $y^2 - a$. Using the fact that the derivative of this function is $2y$, and a little algebra, the above formula reduces to $(1/2)(y + a/y)$, which is the formula we used in section 1.1.7.

[51] In Chapter 2 (section 2.1.3) we shall see that in a language that allows procedures as returned values there is, in principle, no need to include any additional machinery for handling data structures, although Lisp implementations do include such machinery for efficiency reasons. Moreover, we shall see that allowing procedures as returned values enables us to deal with *infinite* data structures via the technique of *stream processing*, to be introduced in Chapter 3.

raises its argument to the 2*n*th power. Write a procedure that takes as inputs a procedure *f* and a positive integer *n* and returns the procedure which is the *n*th repeated application of *f*. For example, your procedure should be able to be used as follows:

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

Exercise 1-34: The idea of *smoothing* a function is an important concept in signal processing. If *f* is a function and *dx* is some small value, then the smoothed version of *f* is the function whose value at a point *x* is the average of *f(x-dx)*, *f(x)*, and *f(x+dx)*. Write a procedure *smooth* that takes as input a procedure that computes *f* and returns a procedure that computes the smoothed *f*. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtained the *n-fold smoothed function*. Show how to easily generate the *n*-fold smoothed function of any given function using *smooth* and *repeated* from exercise 1-33.

Exercise 1-35: Define a procedure *cubic* that can be used together with the Newton's method procedure of section 1.3.4 in expressions of the form

```
(newton (cubic a b c) 1)
```

to approximate roots (starting with an initial guess of 1) to the cubic $x^3 + ax^2 + bx + c$.

Exercise 1-36: Newton's method is an example of a still more general computational strategy known as *iterative improvement*. An iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure *iterative-improve* that takes as arguments an initial guess, a method for telling whether a guess is good enough, and a method for improving a guess. The procedure should return as its value an answer that is good enough. Express Newton's method using the *iterative-improve* procedure. Also show how the fixed-point search (exercise 1-32) can be expressed as an iterative improvement.

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

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

Why do we want compound data in a programming language? For the same reasons that we want compound procedures: to *elevate the conceptual level* at which we can design our programs, to *increase the modularity* of our designs, and to *enhance the expressive power* of our language. Just as the ability to define procedures enables us to deal with processes at a higher conceptual level than that of the primitive operations of the language, so does the ability to construct compound data objects enable us to deal with data at a higher conceptual level than that of the primitive data objects of the language.

Consider, for example, the problem of designing a system to perform arithmetic with rational numbers. For instance, we could imagine an operator `+rat` that takes two rational numbers as arguments and produces their sum. In terms of simple data, a rational number can be thought of as two integers -- a numerator and denominator. So we can design a program in which each rational number is reflected by two integers -- a numerator and a denominator -- and where `+rat` is implemented by two procedures, one that produces the numerator of the sum and one that produces the denominator. But this would be awkward,

because we would then need to explicitly keep track of which numerators correspond to which denominators. In designing a system to perform many operations on many rational numbers, such bookkeeping details would clutter our 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 can manipulate in a way that is consistent with regarding a rational number as a single conceptual unit.

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 below how data abstraction makes programs much easier to design, to maintain, and to modify.

Finally, compound data leads to a real increase in the expressive power of our programming language. Consider for example the idea of forming a "linear combination" $ax + b$. We might like to write a procedure that accepts a, b and x as arguments and returns the value of $ax + b$. This is no problem if the arguments are to be numbers, because we can readily define the procedure

```
(define (linear-combination a b x)
  (+ (* a x) b))
```

But suppose we are not concerned only with numbers. Rather, suppose we would like to express, in procedural terms, the *idea* that one can form linear combinations whenever addition and multiplication are defined -- for rational numbers, complex numbers, polynomials, or whatever. We could express this as a procedure of the form

```
(define (linear-combination a b x)
  (add (mul a x) b))
```

where *add* and *mul* are not the primitive procedures + and *, but rather more complex things that will perform the appropriate operations for whatever kinds of data we pass in as the arguments a, b, and x. The key point is that the only thing that *linear-combination* should need to know about a, b, and x is that the operators *add* and *mul* will perform the appropriate manipulations. From the perspective of the *linear-combination* procedure, it is *completely irrelevant* what a, b, and x 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. For without this, there is no way for a procedure such as *linear-combination* to pass its arguments along to *add* and *mul* without ever having to worry about 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 procedures, the main issue to be addressed is that of *abstraction as a technique for coping with complexity*, and we will see how data abstraction enables us to erect suitable *abstraction barriers* between different parts of a program.

We will see that the key to forming compound data is that a programming language should provide some kind of "glue" that enables one to combine data objects to form more complex data objects. There are many possible kinds of glue. Indeed, we shall discover how to form compound data using no special "data" operations at all, but only procedures. This will further blur the distinction between "procedure" and "data," already made tenuous towards the end of Chapter 1. We will also explore some conventional techniques for representing sequences, trees, and symbolic expressions, with applications to symbolic differentiation, representing sets, and encoding information. Next, we take up the problem of working with data that may be represented differently by different parts of a program. Complex numbers, for example, can be represented in either rectangular or polar form, and for some applications it may be desirable to be able to use *both* representations without sacrificing the ability to work in terms of abstract "complex numbers." This leads to the problem of implementing *generic operators* such as `add` and `mul` alluded to above, which must operate on many different types of data. Maintaining modularity in the presence of generic operators requires erecting more powerful abstraction barriers than can be achieved with simple data abstraction alone, and we introduce *data-directed programming* as a powerful design technique for coping with this complexity.

To illustrate the power of this approach to system design, we close the chapter by applying what we have learned to implement a package for performing symbolic arithmetic on polynomials, where the coefficients of the polynomials can be integers, rational numbers, complex numbers, and even other polynomials. The design of a general-purpose polynomial package, however, is a large enterprise, involving many mathematical and algorithmic as well as system-design issues. Our simple polynomial package will leave much room for further refinements and extensions.

---

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

2.1. Introduction to Data Abstraction

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

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

2.1.1. Example: Arithmetic Operators for Rational Numbers

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

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

`make-rat`	takes two integers *n* and *d* as arguments and returns the rational number whose numerator is *n* and whose denominator is *d*
`numer`	takes a rational number as argument and returns its numerator
`denom`	takes a rational number as argument and returns its denominator

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

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} * \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1}{d_1} \div \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \quad \text{if and only if} \quad n_1 * d_2 = n_2 * d_1$$

We can express these rules as Lisp procedures:

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(define (-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (/rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))

(define (=rat x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Now we have the operators on rational numbers defined in terms of the selector and constructor procedures. But we don't have these defined in our computer. We haven't yet told the computer what a *numer* or a *denom* is, nor have we told it what it must do in order to *make-rat*. What we need is some way to glue together a numerator and a denominator so that they can be combined to form a rational number.

Pairs

To enable us to implement the more concrete level of our data abstraction, our language provides a compound structure called a *pair*, which can be constructed using the primitive procedure *cons*. *Cons* takes two arguments and returns a compound data object that contains the two arguments as parts. Given a pair, we can extract the parts using the primitive procedures *car* and *cdr*.[2] Thus we can use *cons*, *car*, and *cdr* as follows:

```
==>(define x (cons 1 2))
x

==>(car x)
1

==>(cdr x)
2
```

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

```
==>(define x (cons 1 2))
x

==>(define y (cons 3 4))
y

==>(define z (cons x y))
z

==>(car (car z))
1

==>(car (cdr z))
3
```

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

Representing rational numbers

Using pairs, we have a natural way to complete the rational number system. Simply represent a rational number as a pair of two integers, a numerator and a denominator. Then *make-rat*, *numer*, and *denom*, are readily implemented as:

```
(define (make-rat n d) (cons n d))
```

---

[2] The name *cons* stands for "construct." The names *car* and *cdr* relate to the original implementation of Lisp on the IBM 704. That machine had an addressing scheme that allowed one to reference the "address" and "decrement" parts of a memory location. The word *car* stands for "Contents of Address Register" and *cdr* stands for "Contents of Decrement Register."

```
(define (numer x) (car x))

(define (denom x) (cdr x))
```

Also, in order to display the results of our computations, we can choose to print a rational number by printing the numerator, a slash, and the denominator:[3]

```
(define (print-rat x)
  (newline)
  (princ (numer x))
  (princ "/")
  (princ (denom x)))
```

Now we can try our rational number functions:

```
==>(define one-half (make-rat 1 2))
one-half

==>(print-rat one-half)
1/2

==>(define one-third (make-rat 1 3))
one-third

==>(print-rat (+rat one-half one-third))
5/6

==>(print-rat (*rat one-half one-third))
1/6

==>(print-rat (+rat one-third one-third))
6/9
```

As the final example shows, our rational number implementation leaves something to be desired, since it does not reduce rational numbers to lowest terms. We can remedy this by changing *make-rat*. Suppose that we have a *gcd* procedure like the one in section 1.2.5 that produces the greatest common divisor of two integers. Then we can use *gcd* to reduce the numerator and denominator to lowest terms before constructing:

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

Now we have

```
==>(print-rat (+rat one-third one-third))
2/3
```

---

[3] *Print* and *princ* are the Scheme commands for printing data. They are similar, except that *print* starts a new line for printing and terminates its output with a space, while *princ* does not. We implement *print-rat* using *princ* because we want the numerator, slash, and denominator to be printed on the same line. The Scheme command *newline* starts a new line for printing. (Normally, *print* does this automatically.)

as desired. Notice that this modification was accomplished by changing only the constructor *make-rat* without changing any of the procedures that implement the actual operators like *+rat* and **rat*.

Exercise 2-1: In point of fact, this version of *make-rat* is not quite correct, since *make-rat* might be called with negative values for *n* and *d*, and the *gcd* procedure of section 1.2.5 works only with positive integers. Define a better version of *make-rat*, that can handle both positive and negative arguments.

2.1.2. Abstraction Barriers

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

We can envision the structure of our system as shown in figure 2-1. The thick horizontal lines represent *abstraction barriers* that isolate different "levels" of the system. Programs that use rational numbers manipulate them solely in terms of the operators supplied "for public use" by the rational arithmetic package: *+rat*, *-rat*, **rat*, */rat*, and *=rat*. These, in turn, are implemented solely in terms of the constructor and selectors *make-rat*, *numer*, and *denom*, which themselves are implemented in terms of pairs. Finally, the details of how pairs are implemented is *completely irrelevant* to the rest of the rational number package, so long as pairs can be manipulated using *cons*, *car*, and *cdr*. In effect, procedures at each level are the *interfaces* that define the abstraction barriers and connect the different levels.

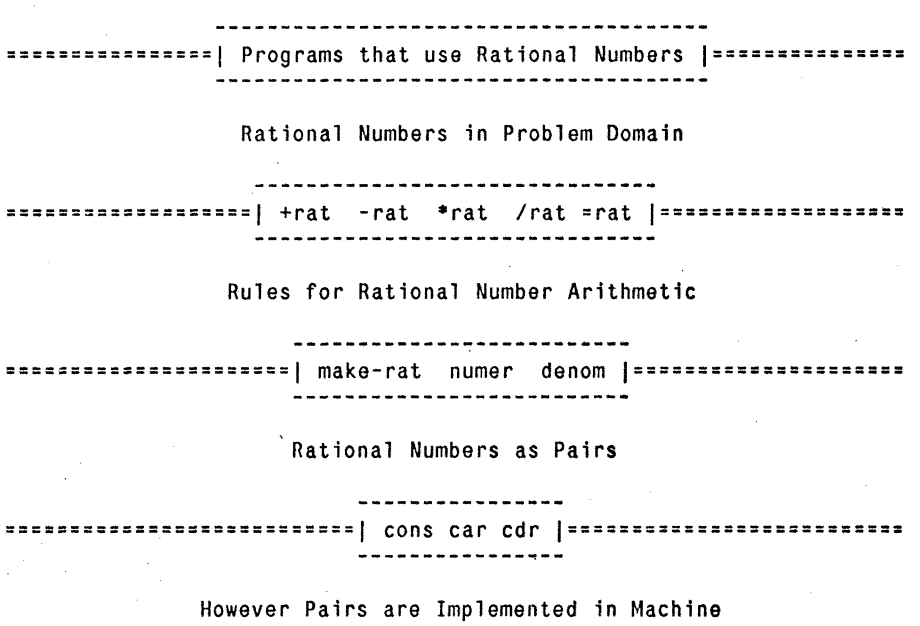


Figure 2-1: Data-abstraction barriers in the rational number package

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

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

```
(define (make-rat n d)
  (cons n d))

(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))

(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

The difference between this implementation and the previous one lies in when we compute the *gcd*. If we examine rational numbers often, then it would be better to compute the *gcd* once in constructing them. If not, we may be better off waiting until the last minute to compute the *gcd* (at access time or even at print time). In any case, when we change from one representation to the other, the operators *+rat*, *-rat*, and so on, do not have to be modified at all.

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

Exercise 2-2: Consider the problem of representing line segments on a two-dimensional plane. Each segment is represented as a pair of points -- a start-point and an end-point. Define a constructor *make-segment* and selectors *start-point* and *end-point* that define the representation of "segments" in terms of "points." Furthermore, a point can be represented as a pair of numbers, the x-coordinate and the y-coordinate. Accordingly, specify a constructor *make-point* and selectors *x-coord* and *y-coord* that define this representation. Finally, using your selectors and constructors, define a procedure *midpoint* that takes a line segment as argument and returns the midpoint (that is, the point whose coordinates are the average of the coordinates of the endpoints).

2.1.3. What is Data?

We began the rational number implementation in section 2.1.1 by implementing the rational number operators *+rat*, *-rat*, and so on, in terms of three unspecified procedures *make-rat*, *numer*, and *denom*. At that point, we could think of the operators being defined in terms of "data objects" -- numerators, denominators, and rational numbers -- whose behavior is specified by the latter three procedures.

But exactly what is this "data"? It is not enough to say that it is "whatever is implemented by the given selectors and constructors." For clearly, any arbitrary three procedures could not serve as an appropriate basis for the rational number implementation. If we think about the problem, we'll see that what we need to guarantee for rational numbers is that, if we construct a rational number *x* from a pair of integers *n* and *d*, then extracting the numerator and denominator of *x* and forming the quotient should yield a rational number equal to *n/d*. In other words, *make-rat*, *numer*, and *denom* are not any three arbitrary procedures. They must satisfy the condition that, for any integers *a* and *b*, if

```
x = (make-rat a b)
```

then

$$\frac{\texttt{(numer x)}}{\texttt{(denom x)}} = \frac{a}{b}$$

In fact, this is the *only* condition that *make-rat*, *numer*, and *denom* must fulfill in order for them to form a suitable basis for a rational number representation. In general, we can think of data as defined by some collection of selectors and constructors, together with specified conditions that these procedures must fulfill in order to be a valid representation.[4]

This point of view can serve to define not only "high level" data objects such as rational numbers, but lower-level objects, as well. Consider, for instance, the notion of a *pair*, which we used in order to define our rational numbers. We never actually said what a pair was, only that the language supplied operators *cons*, *car*, and *cdr* for operating on pairs. But the only thing we need to know about these three operators is that if we glue two objects together using *cons*, then we can retrieve the objects using *car* and *cdr*. That is to say, the operators satisfy the condition:

---

[4]Surprisingly, this idea is very difficult to formulate rigorously. In fact, despite an enormous amount of work in programming language semantics, the notion of a data object, which is so important and pervasive in modern programming practice, does not have a completely satisfactory mathematical treatment. There are two approaches to giving such a treatment. One approach, pioneered by C.A.R. Hoare [22], is known as the method of *abstract models*. It formalizes the "procedures plus conditions" specification as outlined in the rational number example above. Note that the condition on the rational number representation was stated in terms of facts about integers (equality and division). In general, abstract models define new kinds of data objects in terms of previously defined types of data objects. Assertions about data objects can therefore be checked by reducing them to assertions about previously defined data objects. Another approach, introduced by J. Guttag [15], is called *algebraic specification*. It regards the "operators" as elements of an abstract algebraic system, whose behavior is specified by axioms that correspond to our "conditions," and uses the techniques of abstract algebra to check assertions about data objects. Both methods are surveyed in the paper by Liskov and Zilles [29]. These methods work well for simple examples, but become very complex and even break down in complicated situations, and have problems in dealing with "mutable" data objects, such as we will introduce in the next chapter. Resolving these problems is an active area of current research.

- For any objects *x* and *y*, if *z* is *(cons x y)* then *(car z)* is *x* and *(cdr z)* is *y*.

Indeed, we mentioned that these three operators are included as primitives in our language. However, *any* triple of procedures that satisfies the above condition can be used as the basis for implementing pairs.

To make this point in a striking way, we'll show how we could implement *cons*, *car*, and *cdr without using any data structures at all*, but only using procedures! Here are the definitions:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m))))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
```

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

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

The point of this is not that our language works this way (Lisp systems implement pairs directly, for efficiency reasons) but rather that it *could* work this way. The above representation, although obscure, is a perfectly adequate way to represent pairs, since it fulfills the only conditions that pairs need to fulfill. As an interesting sidelight, we see that the ability to manipulate procedures as objects automatically provides the ability to represent compound data. You may regard this as a curiosity for now, but procedural representations of data will play a central role in our programming repertoire. This style of programming is often called *message passing*, and we will be using it as a basic tool in Chapter 3, when we address the issues of modeling and simulation.

Exercise 2-3: Given the following procedural representation, verify that *(car (cons x y))* yields *x*, for any objects *x* and *y*.

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))
```

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

Exercise 2-4: Show that we can represent pairs of non-negative integers using numbers and arithmetic operations only, if we represent the pair *a* and *b* as the integer that is the product $2^a 3^b$. Give the corresponding definitions of the procedures *cons*, *car*, and *cdr*.

Exercise 2-5: In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without *numbers* (!), at least insofar as non-negative integers are concerned, by implementing *0* and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))

(define (1+ n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

Define *1* and *2* directly (not in terms of *zero* and *1+*). (Hint: Use substitution to evaluate *(1+ zero)*).

Give a direct definition of the addition operator + (without introducing any auxiliary procedures).

This representation is known as *Church numerals*, after its inventor Alonzo Church, the logician who invented the λ-calculus.

2.1.4. Example: Interval Arithmetic

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

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

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

Resistance values are usually known only up to some tolerance, guaranteed by the manufacturer of the resistor. For example, if we buy a resistor labelled 6.8 Ohms with a 10% tolerance, we can only be sure that the resistor has a resistance between 6.8 - .68 = 6.12 and 6.8 + .68 = 7.48 Ohms. Thus, if we have a 6.8 Ohm 10% resistor in parallel with a 4.7 Ohm 5% resistor, the resistance of the combination can range from about 2.58 Ohms (if the two resistors are at the lower bounds) to about 2.97 Ohms (if the two resistors are at the upper bounds).

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

Alyssa postulates the existence of an abstract datum called an "interval" that has two parts, a *lower-bound* and an *upper-bound*, which are the endpoints of the interval. She also presumes that given the endpoints of an interval, she can construct the interval using the data

constructor *make-interval*. Alyssa first writes *intadd* for adding two intervals. She reasons that the minimum value the sum could be is the sum of the two lower bounds and the maximum value it could be is the sum of the two upper bounds:

```
(define (intadd x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))))
```

Alyssa also works out the product of two intervals by finding the minimum and the maximum of the products of the bounds and using them as the bounds of the resulting interval:

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

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

```
(define (intdiv x y)
  (intmul x
          (make-interval (/ 1 (upper-bound y))
                         (/ 1 (lower-bound y)))))
```

Exercise 2-6: Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Here is a definition of the constructor *bounds*.

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

Complete the implementation by defining selectors *upper-bound* and *lower-bound*.

Exercise 2-7: Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction procedure, called *intsub*.

Exercise 2-8: The *width* of an interval is the difference between its upper and lower bounds. The width is a measure of the uncertainty of the number specified by the interval. For some arithmetic operators the width of the result of combining two intervals is a function only of the widths of the argument intervals whereas for others, the width of the combination is not a function of the widths of the argument intervals. Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added. Give examples to show that this is not true for multiplication or division.

Exercise 2-9: Ben Bitdiddle, an expert systems programmer, looked over Alyssa's shoulder and commented that it is not clear what it means to divide by an interval that spans zero. Modify Alyssa's code to check for this condition and to signal an error if it occurs.

Exercise 2-10: In passing, Ben also cryptically commented: "By testing the signs of the endpoints to the intervals passed to *intmul*, it is possible to break *intmul* into nine cases, only one of which requires more than two multiplications." Rewrite *intmul* using Ben's suggestion.

After debugging her program, Alyssa showed it to a potential user who complained, as usual, that her program solved the wrong problem. He wanted a program that can could with numbers represented as a center value and an additive tolerance. For example, he wanted to work with intervals such as 3.5±.15 rather than [3.35, 3.65].

Alyssa returned to her desk and fixed this problem by supplying an alternate constructor and alternate selectors as follows:

```
(define (center-width c w)
  (make-interval (- c w) (+ c w)))

(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))

(define (width i)
  (/ (- (upper-bound i) (lower-bound i)) 2))
```

Unfortunately, most of the users really wanted to specify their intervals in terms of an error proportional to the value of the center of the interval, as in the resistor specifications given earlier.

Exercise 2-11: Define a constructor *make-center-percent*, which takes a center and a percentage tolerance and which produces the desired interval. You must also define a selector, *percent*, which produces the correct percentage tolerance for a given interval. The *center* selector is the same as the one shown above.

Exercise 2-12: In real engineering situations we are usually dealing with measurements with only a small uncertainty, measured as the ratio of the width of the interval to the midpoint of the interval. That is why engineers usually specify percentage tolerances on the parameters of devices. Show that, under the assumption of small percentage tolerances, there is a simple formula for the approximate percentage tolerance of the product of two intervals in terms of the tolerances of the factors. You may simplify the problem by assuming that all numbers are positive.

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

$$\frac{1}{1/R_1 + 1/R_2} \quad \text{or} \quad \frac{R_1 * R_2}{R_1 + R_2}$$

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

```
(define (par1 r1 r2)
  (intdiv (intmul r1 r2)
          (intadd r1 r2)))

(define (par2 r1 r2)
  (let ((one (make-interval 1 1)))
       (intdiv one
               (intadd (intdiv one r1)
                       (intdiv one r2)))))
```

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

Exercise 2-13: Demonstrate that the user is right. Investigate the behavior of the system on a variety of expressions. For example, you should make some intervals A and B, and use them in computing the

expressions A/A and A/B. You will get the most insight by using intervals whose width is a small percentage of the center value. Examine the results of the computation in center-percent form.

Exercise 2-14: Eva Lu Ator, another user, has also noticed the problem of different intervals computed by different but algebraically equivalent expressions. She says that a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in a form where no variable that represents an uncertain number is repeated. Thus, she says, *par2* is a "better" program for parallel resistances than *par1*. Is she right? Why?

Exercise 2-15: Explain, in general, why equivalent algebraic expressions may lead to different answers. Can you devise an interval arithmetic package that does not have this problem, or is this task impossible? (Warning: This problem is very difficult.)

2.2. Hierarchical Data

We've seen that pairs provide a primitive "glue" that we can use to construct compound data objects. Figure 2-2 shows a standard way to visualize a pair, in this case, the pair formed by *(cons 1 2)*. In this representation, which is called *box-and-pointer notation*, each object is surrounded by a box. A pair itself is represented as a double box with arrows (also called *pointers*) pointing to the *car* and the *cdr* of the pair.

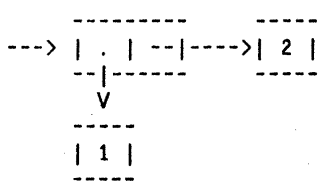


Figure 2-2: Box-and-pointer representation of *(cons 1 2)*.

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

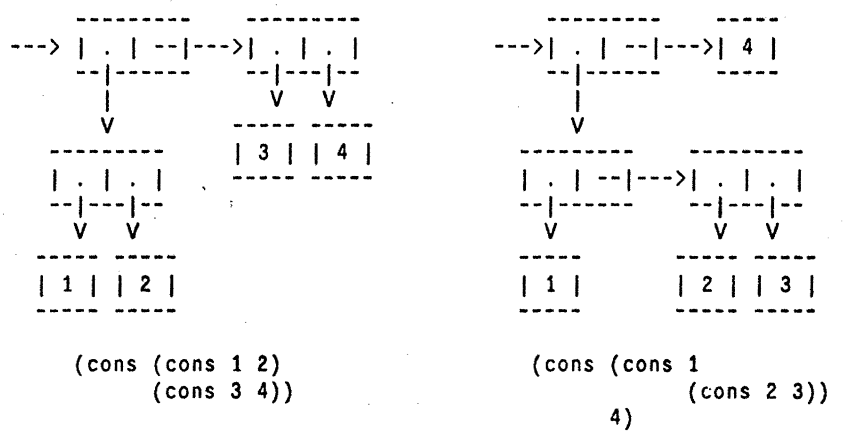


Figure 2-3: Two ways to combine 1, 2, 3, and 4 using pairs.

In general, pairs enable us to represent *hierarchical data* -- data made up of parts, which themselves are made up of parts, and so on. As figure 2-3 indicates, we can use pairs to combine data in many different ways, and we begin this section by exploring some conventional techniques for using pairs to represent sequences and trees. Next, we augment the representational power of our language by introducing *symbolic expressions* -- data whose "elementary parts" can be arbitrary symbols, rather than only numbers. In problem sections, we explore various alternatives for representing *sets* of objects. We will find that, just as a given numerical function can be computed by many different computational processes, there are many ways that 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 also investigate *Huffman encoding* as a clever use of trees to implement efficient codes.

2.2.1. Representing Sequences

One of the useful structures that we can build with pairs is a *sequence* -- an ordered collection of data objects. There, are, of course, many ways to represent sequences in terms of pairs. One particularly straightforward representation is illustrated in figure 2-4, where the sequence 1, 2, 3, 4 is represented as a sequence of pairs. The `car` of each pair is the corresponding item in the sequence, and the `cdr` of the pair is the next pair in the sequence. The `cdr` of the final pair signals the end of the sequence by pointing to a distinguished element, represented in box-and-pointer diagrams as a diagonal line and in Lisp programs as the value of the symbol `nil`.

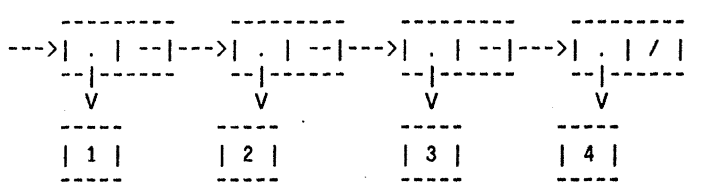


Figure 2-4: The sequence 1, 2, 3, 4 represented as a sequence of pairs.

Observe that this sequence is constructed by nested `cons` operations:

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

Such a sequence of pairs, formed by nested `cons`es, is called a *list*, and Lisp provides a primitive called `list` to help in constructing lists:

```
(list 1 2 3 4)
```

In general,

(`list` $\langle a_1 \rangle$ $\langle a_2 \rangle$ . . . $\langle a_n \rangle$)

is treated by the interpreter as an abbreviation for

(cons <a_1> (cons <a_2> (cons ... (cons <a_n> nil) ...)))

Lisp conventionally prints lists by printing the sequence of elements, enclosed in parentheses. Thus the data object in figure 2-4 is printed as *(1 2 3 4)*:

```
==>(define 1-thru-4 (list 1 2 3 4))
1-thru-4

==>1-thru-4
(1 2 3 4)
```

We can interpret *car* as selecting the first item in the list and *cdr* as selecting the sublist consisting of all but the first item. Nested applications of *car* and *cdr* can be used to extract the second, third, and so on, items in the list.[5] The constructor *cons* can be interpreted as an operation that inserts a new item at the beginning of a list.

```
==>(car 1-thru-4)
1

==>(cdr 1-thru-4)
(2 3 4)

==>(car (cdr 1-thru-4))
2

==>(cons 10 1-thru-4)
(10 1 2 3 4)
```

The value of *nil*, used to terminate the chain of pairs, can be interpreted as a sequence of *no* elements, the so-called *empty-list*. Indeed, "nil" is a contraction of the Latin word for "nothing."[6]

List operations

Accompanying the use of pairs to represent sequences of elements as lists, there are conventional programming techniques for manipulating lists by successively "*cdr*-ing down the sequence" until we reach the empty list. To aid in this, our language includes a primitive predicate *null?*, which tests whether its argument is the empty list. Here is a typical procedure *length*, which returns the number of items in a list:

---

[5] Since nested applications of *car* and *cdr* are cumbersome to write, Lisp provides abbreviations for them. For instance

(cadr <*arg*>) = (car (cdr <*arg*>))

The names of all such procedures start with a "*c*" and end with an "*r*". Each "*a*" between them stands for a *car* operator and each "*d*" for a *cdr* operator, to be applied in the same order in which they appear in the name.

[6] Because of this interpretation, Lisp implementations traditionally regard *( )* and *nil* as different representations of the same data object. Also, recall from section 1.1.6 that *nil* is the value returned by a predicate to indicate "false." This mixing of logical operations with list operations is sometimes convenient, but more often leads to programming errors. People who do not like Lisp regard this feature as one of their favorite things not to like.

```
(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))

==>(define odds (list 1 3 5 7))
odds

==>(length odds)
4
```

The *length* procedure implements the simple recursive plan:

- The length of the empty list is 0.
- The length of any list is 1 plus the length of the *cdr* of the list.

We could also compute *length* in an iterative style:

```
(define (length x)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter x 0))
```

Another useful procedure is *nth*, which takes as arguments a number *n* and a list and returns the *n*th item of the list. It is customary to number the elements of the list beginning with 0. The method for computing *nth* is

- For *n* equal to 0, *nth* should return the *car* of the list.
- Otherwise, the *n*th item of the list is obtained as the *n-1*st item of the *cdr* of the list.

```
(define (nth n x)
  (if (= n 0)
      (car x)
      (nth (- n 1) (cdr x))))

==>(define squares (list 1 4 9 16 25))
squares

==>(nth 3 squares)
16
```

The procedure *append* takes two lists as arguments and combines their elements to make a new list:

```
==>(append squares odds)
(1 4 9 16 25 1 3 5 7)

==>(append odds squares)
(1 3 5 7 1 4 9 16 25)
```

Append is also implemented using a recursive plan: To *append* a list *x* to a list *y*

- If *x* is the empty list, then the result is just *y*.
- Otherwise, *append* the *cdr* of *x* to *y*, and *cons* the *car* of *x* onto the result.

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

Exercise 2-16: Define a procedure *last* that returns the list that contains only the last element of a given (non-empty) list:

```
==>(last squares)
(25)
```

Exercise 2-17: Define a procedure *reverse* that takes a list as argument and returns the same list with the elements in reverse order:

```
==>(reverse squares)
(25 16 9 4 1)
```

Exercise 2-18: One very common pattern of usage is to apply a given procedure to each item in a list by "*cdr*ing down the list, *cons*ing up an answer". For a typical example, given a list of numbers, suppose we want the list of the squares of those numbers. We need a procedure, *square-list* which is used as follows:

```
==>(define 1-thru-4 (list 1 2 3 4))
1-thru-4

==>(square-list 1-thru-4)
(1 4 9 16)
```

Fill in the missing expressions to complete the definition of *square-list*:

```
(define (square-list x)
  (if (null? x)
      nil
      (cons (square <??>)
            (square-list <??>))))
```

Exercise 2-19: Louis Reasoner tried to rewrite *square-list* of exercise 2-18 as an iteration:

```
(define (square-list list)
  (define (iter list answer)
    (if (null? list)
        answer
        (iter (cdr list)
              (cons (square (car list))
                    answer))))
  (iter list nil))
```

Unfortunately, defining *square-list* this way produced the answer list in the reverse order of the one desired. Why?

Louis then tried to fix his bug by interchanging the arguments to *cons*:

```
(define (square-list list)
  (define (iter list answer)
    (if (null? list)
        answer
        (iter (cdr list)
              (cons answer
                    (square (car list))))))
  (iter list nil))
```

This didn't quite work either. Explain.

Exercise 2-20: We can improve on exercise 2-18 by following the method of section 1.3 to introduce a higher order procedure that expresses the general operation of applying a procedure to every item in a list, and returning the list of results. This procedure is traditionally called *mapcar*, that is used as follows:

```
==>(mapcar square 1-thru-4)
(1 4 9 16)

==>(mapcar 1+ 1-thru-4)
(2 3 4 5)
```

Give an appropriate definition of *mapcar*.[7]

Exercise 2-21: Consider the change-counting program of section 1.2.2. It would be nice to be able to change the currency used by the program easily, so that we could compute the number of ways to change an British Pound, for example. As the program is written, it has the knowledge of the currency partly distributed into the procedure *first-denomination* and partly into the procedure *count-change* (which knows that there are 5 different kinds of U.S. coins). It would be nicer to be able to supply a list of coins which could be used for making change.

We want to rewrite the essential procedure, *cc*, so that its second argument, *kinds-of-coins*, is a list of coins which may be used, rather than an integer specifying which coin to use. We could then have lists which defined each kind of currency:

```
(define us-coins (list 50 25 10 5 1))

(define uk-coins (list 50 20 10 5 2 1 .5))
```

We could then call *cc* as follows:

```
==>(cc 100 us-coins)
292
```

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

```
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (no-more? kinds-of-coins)) 0)
        (else (+ (cc (- amount (first-denomination kinds-of-coins))
                     kinds-of-coins)
                 (cc amount
                     (except-first-denomination kinds-of-coins))))))
```

a. Define the procedures *first-denomination*, *except-first-denomination*, *no-more?* in terms of the primitive operators on list structures.

b. Does the order of the list *kinds-of-coins* effect the answer produced by *cc*? Why or why not?

---

[7] *Mapcar* barely hints at the expressive power to be gained by combining higher order procedures with hierarchical data. We will have much more to say about this when we study stream processing in Chapter 3 (section 3.4.2).

2.2.2. Representing Trees

The representation of sequences in terms of *cons* pairs generalizes naturally to enable us to represent sequences whose *elements* may *themselves* be sequences. For example, we can regard the object

```
==>(cons (list 1 2) (list 3 4))
((1 2) 3 4)
```

as a list of three items, the first of which is itself the list *(1 2)*. And indeed, this is suggested by the form in which the result is printed by the interpreter. Figure 2-5 shows the representation of this structure in terms of pairs.

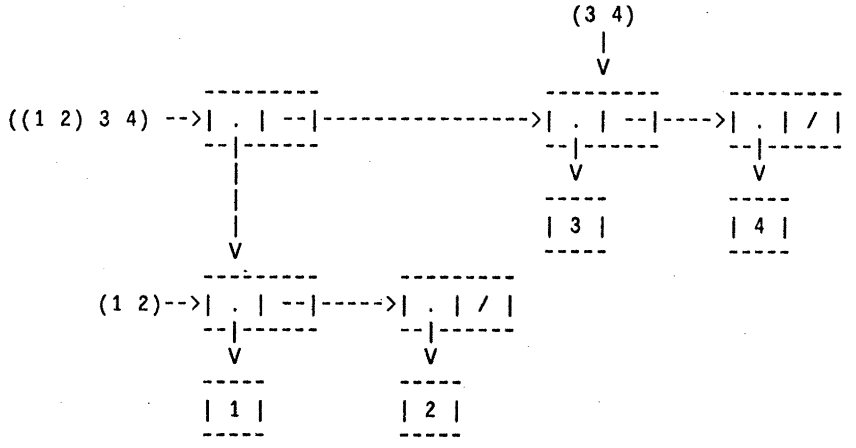


Figure 2-5: Structure formed by *(cons (list 1 2) (list 3 4))*.

Another way to interpret sequences whose elements are sequences is as *trees*. The elements of the sequence are the branches of the tree, and elements which are themselves sequences are interpreted as subtrees. Figure 2-6 shows the structure in figure 2-5 interpreted as a tree.

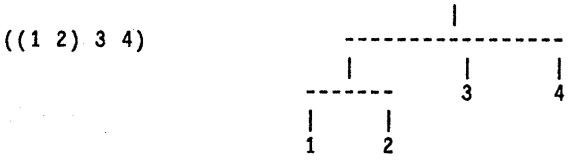


Figure 2-6: The pair structure in figure 2-5, interpreted as a tree.

Exercise 2-22: Suppose we evaluate the combination

```
(list 1 (list 2 (list 3 4)))
```

Give

a. the result printed by the interpreter

b. the corresponding box-and-pointer structure

c. the interpretation of this as a tree, as in Figure 2-6

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 elementary so-called *atomic* data that forms the leaves of the tree. To aid in writing recursive procedures on trees, Lisp provides the primitive predicate `atom?`, which tests whether its argument is atomic (i.e., not a pair).

As an example, compare the `length` procedure of section 2.2.1 with the procedure `countatoms`, which returns the total number of atoms at all levels of a tree:

```
==>(define x (cons (list 1 2) (list 3 4)))
x

==>(length x)
3

==>(countatoms x)
4

==>(list x x)
(((1 2) 3 4) ((1 2) 3 4))

==>(length (list x x))
2

==>(countatoms (list x x))
8
```

To implement `countatoms`, recall the recursive plan for computing `length`. There is a reduction step:

- Length of a list *x* is 1 plus length of `(cdr x)`

that we apply successively until we reach the base case:

- Length of `nil` is 0

`Countatoms` is similar. The value for `nil` is the same:

- Countatoms of `nil` is 0

But in the reduction step, where we strip off the `car` of the list, we must take into account that `(car x)` may *itself* be a list whose atoms we need to count. So the appropriate reduction step is

- `(countatoms x) = (countatoms (car x)) + (countatoms (cdr x))`

Finally, if we keep taking successive `car`s of `car`s we eventually get down to atoms. So we need another base case

- Countatoms of an atom is 1.

Here is the complete procedure:[8]

```
(define (countatoms x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (countatoms (car x))
                 (countatoms (cdr x))))))
```

Exercise 2-23: Modify your *reverse* procedure of exercise 2-17 to produce a procedure *deep-reverse* 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, that is

```
==>(define x (cons (list 1 2) (list 3 4)))
x

==>x
((1 2) 3 4)

==>(reverse x)
(4 3 (1 2))

==>(deep-reverse x)
(4 3 (2 1))
```

Exercise 2-24: Write a procedure *fringe*, that takes a list as argument and returns a list whose elements are all the atoms appearing in the original list or any of its sublists, arranged in left to right order. (That is, given a tree, *fringe* returns the list of leaves of the tree.) For example,

```
==>(define x (cons (list 1 2) (list 3 4)))
x

==>(fringe x)
(1 2 3 4)

==>(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

Exercise 2-25: A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using *list*):

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

A branch is constructed from a *length* (which must be a number) and a *supported-structure*, which may be either a number (representing a simple weight) or another mobile:

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

a. Supply the corresponding selectors *left-branch* and *right-branch*, which return the branches of a mobile, and *branch-length* and *branch-structure*, which return the components of a branch.

b. Using your selectors, define a procedure *total-weight* that returns the total weight of a mobile.

c. A mobile is said to be *balanced* if

---

[8] Notice that the order of the first two clauses in the *cond* matters, since *nil* satisfies both *null?* and *atom?*.

- The torque applied by its top-left branch is equal to that applied by its top-right branch. That is, the length of the left rod multipled by the weight hanging from that rod is equal to the corresponding product for the right side; and
- Each of the sub-mobiles 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 now

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

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

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

2.2.3. Symbolic Expressions; The Need for QUOTE

All of the compound data we have used so far was constructed ultimately from numbers. Now we extend the representational capability of our language by introducing the ability to work with arbitrary *symbols* as data. If we form compound data using not only numbers as atoms, but rather arbitrary symbols (alphanumeric character strings) we obtain the class of data called *symbolic expressions*. Some examples of symbolic expressions are

```
(a b c d)
(23 45 17)
((Bob 20) (Jane 18) (Jim 25))
(* (+ 23 45) (+ x 9))
(define (fact x) (cond ((= x 0) 1) (else (* x (fact (- x 1))))))
```

In order to form symbolic expressions we need a new element in our language -- the ability to *quote* a data object. To see this, suppose we want to construct the list `(a b)`. We can't accomplish this by

```
(list a b)
```

because the interpreter will think that we mean to combine in a list the *value* of *a* and the *value* of *b* rather than the symbols themselves.

This problem is well known in the context of natural languages, where words and sentences may be regarded either as semantic entities or as character strings (syntactic entities). The common practice in natural languages is to use quotation marks to indicate that a word or a sentence is to be treated literally as a string of characters. For instance, the first letter of "John" is clearly "J". If we tell somebody "say your name aloud" we expect to hear that person's name. However, if we tell somebody "say 'your name' aloud" we expect to hear the words "your name". Note that we are forced to nest quotation marks to describe what

somebody else might say.[9]

We can follow this same practice to identify lists and atoms that are to be treated as data objects rather than as expressions to be evaluated. Our format for quoting differs, however, from that used by natural languages, in that we place a quotation mark (traditionally, the single quote symbol ') only at the beginning of the object to be quoted.[10]

Now we can distinguish between symbols and their values:

```
==>(define a 1)
a

==>(define b 2)
b

==>(list a b)
(1 2)

==>(list 'a 'b)
(a b)

==>(list 'a b)
(a 2)
```

Quotation also allows us to type in compound objects, using the conventional printed

---

[9]Allowing *quote* in a language wreaks havoc with the ability to reason simply about the language, because it destroys the notion that "equals can be substituted for equals." For example, three is one plus two, but the word "three" is not "one plus two." More significantly, we will see that quote gives us a way to build expressions that manipulate other expressions (for instance, when we write an interpreter in Chapter 4). And allowing statements in a language that talk about other statements in that language makes it very difficult to maintain any coherent principle of what "equals can be substituted for equals" should mean. As an example, if we know that the evening star is the morning star then, from the statement "the evening star is Venus" we can deduce "the morning star is Venus". However, given that "John knows that the evening star is Venus" we cannot infer that "John knows that the morning star is Venus".

[10]We can get away with this in Lisp syntax because we rely on the blanks and parentheses to delimit objects. Thus the meaning of the single quote character is to quote the next object. Notice that single quote is different from the double quote characters we have been using to enclose character strings to be printed. While single quote can be used to denote lists or symbols, double quote is used only with character strings. In Scheme, the only use for character strings is as items to be printed.

representation for lists:[11]

```
==>(car '(a b c))
a

==>(cdr '(a b c))
(b c)
```

One additional primitive used in manipulating symbolic expressions is *eq?*, which takes two atomic symbols as arguments and tests whether they are the same.[12] Using *eq?*, we can implement a useful procedure called *memq*. This takes two arguments, an atom and a list. If the atom is not contained in the list (i.e., is not *eq?* to any item in the list) then *memq* returns *nil*. Otherwise, it returns the sublist of the list beginning with the first occurrence of the atom:

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

==>(memq 'apple '(1 2 3 4 5 6))
nil

==>(memq 'apple '(x (apple sauce) y apple pair))
(apple pair)
```

Exercise 2-26: What would the interpreter print in response to evaluating each of the following expressions?

```
(list 'a 'b 'c)

(list (list 'george))

(cdr '((x1 x2) (y1 y2)))

(cadr '((x1 x2) (y1 y2)))

(atom? (car '(a short list)))
```

---

[11] Our use of the quotation mark, strictly speaking, violates the general rule that all expressions in our language should be represented as combinations. We can recover this consistency by introducing a special form *quote*, which serves the same purpose as the quotation mark. Thus we would type *(quote a)* instead of *'a*, and *(quote (a b c))* instead of *'(a b c)*. In fact, this is precisely how the interpreter works. The quotation mark is just a single character abbreviation for wrapping the next complete expression with "*(quote ⟨expression⟩)*". This is important because it maintains the principle that *any* expression seen by the interpreter can be manipulated as a data object. For instance, we could construct the expression

```
(car '(a b c)) = (car (quote (a b c)))
```

as

```
(list 'car (list 'quote '(a b c)))
```

[12] We can consider two symbols to be "the same" if they consist of the same characters in the same order. Such a definition skirts a deep issue, which we are not yet ready to address: the meaning of "sameness" in a programming language. We will return to this in Chapter 3 (section 3.1.2).

```
(memq 'red '((red shoes) (blue socks)))

(memq 'red '(red shoes blue socks))
```

Exercise 2-27: Two lists are said to be *equal?* if they contain equal elements arranged in the same order. For example,

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

is true, while

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

is false.

More precisely, we can define *equal?* recursively in terms of the basic *eq?* equality of atoms by saying that *a* and *b* are *equal?* if either they are both atoms and the atoms are *eq?*, or else they are both lists such that *(car a)* is *equal?* to *(car b)* and *(cdr a)* is *equal?* to *(cdr b)*. Using this idea, implement *equal?* as a procedure.

Exercise 2-28: Eva Lu Ator types to the interpreter the expression

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

To her surprise, the interpreter prints back *quote*. Explain. What would be printed in response to

```
(cdddr '(this list contains '(a quote)))
```

2.2.4. Example: Symbolic Differentiation

As an illustration of symbol manipulation and a further illustration of data abstraction, let's consider the problem of designing a procedure that performs symbolic differentiation of algebraic expressions. We would like the procedure to take as arguments an expression and a variable and to return the derivative of the expression with respect to the variable. For example, if the arguments to the procedure are $ax^2 + bx + c$ and x then the procedure should return $2ax + b$. Symbolic differentiation is of special historical significance in Lisp. It was one of the motivating examples behind the development of a computer language for symbol manipulation. Furthermore, it marked the beginning of the line of research that led to the development of powerful systems for symbolic mathematical work, which are currently being used by a growing number of applied mathematicians and physicists.

In developing the symbolic differentiation program, we will follow the same strategy of data abstraction as we did with the rational number system of section 2.1.1. That is to say, we will first define a differentiation algorithm that operates on abstract objects like "sums," "products," and "variables" without worrying about how these are to be represented. Only afterwards will we address the representation problem.

The differentiation program with abstract data

In order to keep things simple, we'll consider a very simple symbolic differentiation program that handles expressions that are built up using only the operations of addition and multiplication with two arguments. Differentiation of any such expression can be carried out by applying the following reduction rules:

$dc/dx = 0$ for c a constant or a variable different from x

$dx/dx = 1$

$$d(u+v)/dx = du/dx + dv/dx$$

$$d(uv)/dx = u(dv/dx) + v(du/dx)$$

Observe that the latter two rules are recursive in nature. That is, to obtain the derivative of a sum, we first find the derivatives of the addends and add them. Each of the addends may in turn be an expression that needs to be decomposed. Decomposing into smaller and smaller pieces will eventually produce pieces that are either constants or variables, whose derivatives will be either 0 or 1.

To embody these rules in a procedure we indulge in a little "wishful thinking" just as we did in designing the rational number implementation. If we had a means for representing algebraic expressions, we should be able to tell the type of an expression: Is it a sum? a product? a constant? a variable? We should be able to get the parts of an expression (e.g., for a sum we want to be able to extract its addend and augend) and we want to construct expressions from parts. So let us assume that we already have procedures to implement the following selectors, constructors, and predicates:

`(constant? e)` Is *e* a constant?

`(variable? e)` Is *e* a variable?

`(same-variable? v1 v2)`
Are *v1* and *v2* the same variable?

`(sum? e)` Is *e* a sum?

`(product? e)` Is *e* a product?

`(addend e)` Addend of the sum *e*.

`(augend e)` Augend of the sum *e*.

`(multiplier e)` Multiplier of the product *e*.

`(multiplicand e)`
Multiplicand of the product *e*.

`(make-sum a1 a2)`
Construct the sum of *a1* and *a2*.

`(make-product m1 m2)`
Construct the product of *m1* and *m2*.

Using these operators, we can express the differentiation rules as the following procedure:

```
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))))
```

This procedure incorporates the complete differentiation algorithm. Since it is expressed in terms of abstract data, it will work no matter how we choose to represent algebraic expressions, as long as we design a proper set of selectors and constructors. This is the issue we must address next.

Representing algebraic expressions

We can imagine many ways to use list structure to represent algebraic expressions. For example, we could use lists of symbols that mirror the usual algebraic notation, representing $ax + b$ as the list `(a * x + b)`. However, one especially straightforward choice is to use the same parenthesized prefix notation that Lisp uses for combinations; that is, $ax+b$ is represented as `(+ (* a x) b)`. Then our data representation for the differentiation problem is as follows:

- The constants are numbers, identified by the primitive predicate *number?*.:

```
(define (constant? x) (number? x))
```

- The variables are symbols, identified by the primitive predicate *symbol?*:

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

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

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

- Sums and products are constructed as lists:

```
(define (make-sum a1 a2) (list '+ a1 a2))

(define (make-product m1 m2) (list '* m1 m2))
```

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

```
(define (sum? x)
  (and (not (atom? x)) (eq? (car x) '+)))
```

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

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

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

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

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

```
(define (product? x)
  (and (not (atom? x)) (eq? (car x) '*)))
```

- The multiplier is the second item of the product list:

```
(define (multiplier p) (cadr p))
```

- The multiplicand is the third item of the product list:

```
(define (multiplicand p) (caddr p))
```

So we need only combine these with the algorithm as embodied by *deriv*, and we should have a working symbolic differentiation program. And indeed the program works, sort of. Let us look at some examples of its behavior:

```
==>(deriv '(+ x 3) 'x)
(+ 1 0)

==>(deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))

==>(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
   (* (+ (* x 0) (* y 1))
      (+ x 3)))
```

The program produces answers that are algebraically equivalent to the correct answers. However, they are unsimplified. It is true that

$$d(xy)/dx = x \cdot 0 + 1 \cdot y$$

but we would like the program to know that $x \cdot 0 = 0$, $1 \cdot y = y$, and $0 + y = y$. So the answer for the second example should have been simply *y*. As the third example shows, this becomes a serious problem when the expressions are complex.

Our problem is much like the one we encountered with the rational number implementation: we haven't reduced answers to lowest form. Remember that, in order to accomplish the rational number reduction, we needed to change only the constructors and selectors of the implementation. We can adopt a similar strategy here. We won't change *deriv* at all. Instead, we'll change *make-sum* so that if one of the summands is 0, then *make-sum* will return the other summand. Also, if both summands are numbers, *make-sum* will add them and return their sum.

```
(define (make-sum a1 a2)
  (cond ((and (number? a1) (= a1 0)) a2)
        ((and (number? a2) (= a2 0)) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
```

Similarly, we'll change *make-product* to build in the rules that 0 times anything is zero and 1 times anything is the thing itself:

```
(define (make-product m1 m2)
  (cond ((and (number? m1) (= m1 0)) 0)
        ((and (number? m2) (= m2 0)) 0)
        ((and (number? m1) (= m1 1)) m2)
        ((and (number? m2) (= m2 1)) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

Here is how this version works on our three examples:

```
==>(deriv '(+ x 3) 'x)
1

==>(deriv '(* x y) 'x)
y

==>(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

Although this is quite an improvement, the third example shows that there is still a long way to go before we get a program that puts expressions in a form that we might agree is "simplest." The problem of algebraic simplification is quite complex because, among other reasons, a form that may be simplest for one purpose may not be for another.

Exercise 2-29: Suppose we want to modify the differentiation program so that is works with algebraic infix notation, rather than with prefix notation. Since the differentiation program is defined in terms of abstract data, we can modify it to work with different representations of expressions *solely* by changing the predicates, selectors, and constructors that define the representation of the algebraic expressions on which the differentiator is to operate.

1. Show how to do this in order to differentiate algebraic expressions presented in infix form, such as `(x*(x+3))`. This is not difficult if we assume that + and * always take two arguments and that expressions are fully parenthesized.
2. The problem becomes substantially harder if we allow standard algebraic notation, such as `x+3*(x+y)`, which drops unnecessary parentheses and assumes that multiplication is done before addition. Can you design appropriate predicates, selectors and constructors for this notation such that our derivative program still works?

Exercise 2-30: Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule

$$d(u^n)/dx = nu^{n-1}(du/dx)$$

by adding a new clause to the `deriv` program and extending the interface to the data by defining appropriate procedures `exponentiation?`, `base`, `exponent`, and `make-exponentiation`. (You may use the symbol ** to denote the exponentiation operator.)

2.2.5. Example: Representing Sets

In the previous sections, we built representations for two kinds of compound data objects, rational numbers and algebraic expressions. In one of these examples, we had the choice of simplifying (reducing) the expressions at either construction or selection time. But other than that, the choice of a representation for these structures in terms of lists was straightforward. On the other hand, when we turn to the problem of representing *sets*, we'll find that the choice of representation is not so obvious. Indeed, we shall see that there are a number of possible

representations, which differ significantly from one another in several ways.

Informally, a set is simply a collection of distinct objects. To give a more precise definition we can employ the method of data abstraction. That is, we define "set" by specifying the operators that are to be used on sets. These operators are: *union-set*, *intersection-set*, *element-of-set?*, and *adjoin-set*. *Element-of-set?* is a predicate that determines whether a given element is a member of a set. *Adjoin-set* takes an object and a set as arguments, and returns a set that contains the elements of the original set and also the adjoined element. *Union-set* computes the union of two sets, which is the set containing each element that appears in either argument. *Intersection-set* computes the intersection of two sets, which is the set containing only elements that appear in both arguments. We will use the empty list to represent the empty set. From the point of view of data abstraction, we are free to design any representation that implements these operators in a way consistent with the interpretations given above.[13]

Sets as unordered lists

One way to represent a set is as a list of its elements, where no element appears more than once. The empty set is represented by the empty list. In this representation, *element-of-set?* is similar to the procedure *memq* of section 2.2.3:

```
(define (element-of-set? x set)
  (cond ((null? set) nil)
        ((equal? x (car set)) t)
        (else (element-of-set? x (cdr set)))))
```

Using this, we can write *adjoin-set*. If the object to be adjoined is already in the set we just return the set. Otherwise, we use *cons* to add the object to the list that represents the set:

---

[13] If we want to be more formal, we can specify "consistent with the interpretations given above" to mean that the operators satisfy a collection of rules such as:

- For any set *S* and any object *x*

  `(element-of-set? x (adjoin-set x S))`

  is true (informally: "adjoining an object to a set produces a set that contains the object").
- For any sets *S* and *T*, and any object *x*,

  `(element-of-set? x (union-set S T))`

  is equal to

  `(or (element-of-set? x S) (element-of-set? x T))`

  (informally: "the elements of *(union S T)* are the elements that are in *S* or in *T*").
- For any object *x*

  `(element-of-set? x '())`

  is *nil* (informally: "no object is an element of the empty set")
- and so on.

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

For *intersection-set*, we can use a recursive strategy: Suppose we have already formed the intersection of the *set2* and the *cdr* of *set1*. Then we only need to decide whether or not to include the *car* of *set1* in this. But this depends on whether *(car set1)* is also in *set2*. Here is the resulting procedure:

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) nil)
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

Exercise 2-31: Give the analogous implementation of *union-set*.

In designing a representation, one of the issues we should be concerned with is how efficient the representation is. Let's consider the time required by our set operations. Since most of these use *element-of-set?*, the speed of this operation has a major impact on the efficiency of the set implementation as a whole. Now in order to check whether an object is a member of a set, *element-of-set?* must scan the entire set. (In the worst case, the object turns out not to be in the set). Hence if the set has n elements, *element-of-set?* might take up to n steps. So the time required grows as $O(n)$. The time required by *adjoin-set*, which uses this operation, also grows as $O(n)$. For *intersection-set*, which does an *element-of-set?* check for each element of *set1*, the time required grows as the product of the sizes of the sets involved, or $O(n^2)$ for two sets of size n. The same will be true of *union-set*.

Exercise 2-32: We specified that a set would be represented as a list with no duplicate elements. Now suppose we allowed duplicates. For instance, the set {1,2,3} could be represented as the list *(2 3 2 1 3 2 2)*. Design procedures *element-of-set?*, *adjoin-set*, *union-set*, and *intersection-set* that operate on this representation. How does the efficiency of each compare with the corresponding procedure for the non-duplicate representation? Are there applications for which you would use this representation in preference to the non-duplicate one?

Ordered lists

One way to speed up our set operations is to change the representation so that the set elements are listed in increasing order. To do this, we need some way to compare two objects so that we can say which is bigger. For example, we could compare symbols lexicographically, or agree on some method for assigning a unique number to an object and then compare the elements by comparing the corresponding numbers. To keep our discussion simple, we'll consider only the case where the set elements are themselves numbers, so that we can compare elements using $>$ and $<$. We'll represent a set of numbers by listing its elements in increasing order. While our first representation above allowed us to represent the set {1,3,6,10} by listing the elements in any order, our new representation allows only the list *(1 3 6 10)*.

One advantage of ordering shows up in *element-of-set?*. In checking for the presence

of an item, we no longer have to scan the entire set. If we reach a set element that is larger than the item we are looking for, then we know that the item is not in the set:

```
(define (element-of-set? x set)
  (cond ((null? set) nil)
        ((= x (car set)) t)
        ((< x (car set)) nil)
        (else (element-of-set? x (cdr set)))))
```

How much time does this save? In the worst case, the item we are looking for may be the largest one in the set, so the number of steps is the same as for the unordered representation. On the other hand, if we search for items of many different sizes we can expect that sometimes we will be able to stop searching at a point near the beginning of the list and that other times we will still need to examine most of the list. On the average we should expect to have to examine about half the number of items in the set in a typical search. So the average time required will be about $n/2$. This is still $O(n)$ growth, but it does save us on the average a factor of 2 over the previous implementation.

Exercise 2-33: Give an implementation of *adjoin-set* using the ordered representation. By analogy with *element-of-set?* show how to take advantage of the ordering to produce a procedure that requires on the average about half as many recursions as with the unordered representation. Compare them by counting ***conses***.

We obtain a more impressive speedup when we consider *intersection-set*. In the unordered representation, this operation required time $O(n^2)$, because we performed a complete scan of *set2* for each element of *set1*. But with the ordered representation, we can use a cleverer method: Begin by comparing the initial elements, *x1* and *x2*, of the two sets. If $x1 = x2$ then that gives an element of the intersection, and the rest of the intersection is the intersection of the *cdr*'s of the two sets. Suppose, however, that *x1* is less than *x2*. Since *x2* is the smallest element in *set2* we can immediately conclude that *x1* cannot appear anywhere in *set2* and hence is not in the intersection. Hence, the intersection is equal to the intersection of *set2* with the *cdr* of *set1*. Similarly, if *x2* is less than *x1*, then the intersection is given by the intersection of *set1* with the *cdr* of *set2*. Here is the procedure:

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      nil
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1
                     (intersection-set (cdr set1)
                                       (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((< x2 x1)
               (intersection-set set1
                                 (cdr set2)))))))
```

To estimate the time required by this process, observe that at each step we reduce the intersection problem to computing intersections of smaller sets: removing the first element from *set1* or *set2* or both. So the number of steps required is at most the sum of the sizes of *set1* and *set2*, rather than the product of the sizes as with the unordered representation.

This is $O(n)$ growth rather than $O(n^2)$, and is a considerable speedup, even for sets of moderate size.

Exercise 2-34: Give the analogous $O(n)$ implementation of `union-set` for sets implemented as ordered lists.

Sets as binary trees

We can do better than the ordered list representation by arranging the set elements in the form of a tree. Each node of the tree holds one element of the set, called the "entry" at that node, and a link to each of two other (possibly empty) nodes. The "left" link points to elements smaller than the one at the node, and the "right" link to those elements that are greater than the element at the node. Figure 2-7 shows some trees that represent the set {1,3,5,7,9,11}. Note that the same set may be represented by a tree in a number of different ways. The only thing we require for a valid representation is that all elements in the left subtree be smaller than the node entry and that all elements in the right subtree be larger.

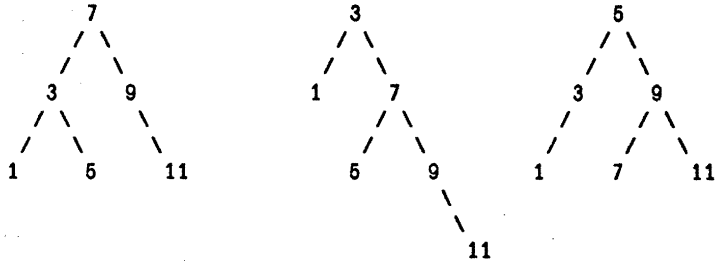


Figure 2-7: Various binary trees that represent the set {1,3,5,7,9,11}.

The advantage of the tree representation is this: Suppose we want to check if a number x is contained in a set. We begin by comparing x with the entry in the top node. If x is less than this, we know that we need only search the left subtree, while if x is greater, we need only search the right subtree. Now if the tree is "balanced," each of these subtrees will be about half the size of the original. So in one step, we have reduced the problem of searching a tree of size n to searching a tree of size $n/2$. Since the size of the tree is halved at each step, we should expect that the number of steps needed to search a tree of size n grows as $O(\log n)$.[14] For large sets, this will be a significant speedup over the previous representations.

We can represent trees by using lists. Each node will be a list of three items: the entry at the node, the left subtree, and the right subtree. A left or right subtree of `nil` will indicate that there is no subtree connected there. We can describe this representation by the following

[14] Halving the size of the problem at each step is the distinguishing characteristic of logarithmic growth, as we saw with the fast exponentiation algorithm of section 1.2.4 and the half-interval search method of section 1.3.3.

procedures:[15]

```
(define (entry tree) (car tree))

(define (left-branch tree) (cadr tree))

(define (right-branch tree) (caddr tree))

(define (make-tree entry left right)
  (list entry left right))
```

Now we can write the *element-of-set?* procedure using the strategy described above:

```
(define (element-of-set? x set)
  (cond ((null? set) nil)
        ((= x (entry set)) t)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Adjoining an item to a set is implemented similarly, and also requires $O(\log n)$ steps. To adjoin an item x, we compare x with the node entry to determine whether x should be added to the right or to the left branch, and having adjoined x to the appropriate branch, we piece this together with the original entry and the other branch. If x is equal to the entry, we just return the original set. And if we are asked to adjoin x to an empty tree, we generate a tree that has x as the entry and null right and left branches. Here is the procedure:

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x nil nil))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x
                                (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x
                                (right-branch set))))))
```

For large sets, the tree representation is much more efficient than the ordered or unordered lists for searching and adjoining new elements. For computing intersections, however, it turns out that there is no general way to proceed other than to use the same strategy as we did in the unordered list representation. That is, for each element in *set1*, we scan to see if this is in *set2*, and, if so, adjoin it to an intersection set that we accumulate. Since the search

---

[15] Observe that we are representing sets in terms of trees, and representing trees in terms of lists -- in effect, a data abstraction built upon a data abstraction. We can regard the procedures *entry*, *left-branch*, *right-branch*, and *make-tree* as a way of isolating the abstraction of "a binary tree" from the particular way we might wish to represent such a tree in terms of lists.

requires time roughly equal to the logarithm of the number of items in *set2*, and we must do this operation for each element in *set1*, the total time required grows as the size of *set1* times the logarithm of the size of *set2*, or $O(n \log n)$ if the two sets have comparable size. This is still much better than for the unordered list representation, but not quite as good as the ordered list representation. Since a typical set implementation is likely to do much more searching than intersection, the tree representation is usually to be preferred.

Exercise 2-35: How does the tree representation of sets compare with other representations on set union problems?

Finally, we mention an additional problem with the tree implementation. The claim that searching the tree can be performed in logarithmic time rests on the assumption that the tree is "balanced," i.e., that the left and right subtrees of every tree have approximately the same number of elements, so that each subtree contains about half the elements of its parent. But how can we be certain that the trees we construct will be balanced? Even if we start with a balanced tree, adding elements with *adjoin-set* may produce an unbalanced result. Since the position of a newly adjoined element depends on how the element compares with the items already in the set, we could expect that if we add elements "randomly," the tree will tend to be balanced on the average. But this is not a guarantee. For example, if we start with an empty set and adjoin the numbers 1 through 7 in sequence, we end up with the highly unbalanced tree shown in figure 2-8. In this tree, all the left subtrees are empty, and so it has no advantage over a simple sorted list. One way to solve this problem is to define an operation that transforms an arbitrary tree into a balanced tree with the same elements. Then we can perform this transformation after every few *adjoin-set* operations to keep our set in balance. There are also other ways to solve this problem, most of them involving designing new data structures, for which searching and insertion both can be done in $O(\log n)$ steps.[16]

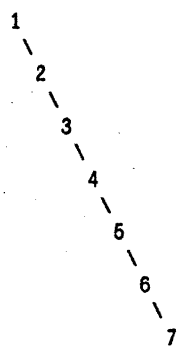


Figure 2-8: Unbalanced tree produced by adjoining 1 through 7 in sequence.

Sets and information retrieval

We have examined options for using lists to represent sets. This illustrated how the choice of representation for a data object can have a large impact on the performance of the programs that use the data. But another reason for concentrating on sets is that the

[16] Examples of such structures include "heaps," "2-3 trees," "AVL-trees." There is a large literature on data structures which is devoted to this problem.

techniques discussed here appear again and again in applications involving information retrieval.

Consider a large data base that consists of a large number of individual records -- for example, personnel files for a company, or the transactions in an accounting system. A typical data management system spends a large amount of time accessing or modifying the data in the records and therefore requires an efficient method for accessing records. To do this, we identify a part of each record to serve as an identifying *key*. A key can be anything at all, so long as the key uniquely identifies the record. For a personnel file, it might be an employee social security number. For an accounting system, it might be a transaction number. Whatever the key is, when we define the record as a data structure, we should include a *key* selector procedure, that retrieves the key associated with a given record.

Now we represent the data base as a set of records. To locate the record with a given key, we use a procedure *lookup*, which takes as argument a key and a data base and returns the record that has that key, or *nil* if there is no such record. *Lookup* is implemented in almost the same way as *element-of-set?*. For example, if the set of records is implemented as an unordered list, then we could use:

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) nil)
        ((equal? given-key (key (car set-of-records)))
         (car set-of-records))
        (else (lookup given-key (cdr set-of-records)))))
```

Of course, we've seen that there are better ways to represent large sets than as unordered lists. Information retrieval systems in which records have to be "randomly accessed" are typically implemented using a tree-based method such as the binary tree representation discussed previously. In designing such a system, the methodology of data abstraction can be a great help. The designer can create an initial implementation using a simple, straightforward representation such as unordered lists. This will be unsuitable for the eventual system, but it can be useful in providing a "quick and dirty" data base with which to test the rest of the system. Later on, the data representation can be modified to be more sophisticated. And if the data base is accessed in terms of abstract selectors and constructors, this change in representation will not require making any changes to the rest of the system.

Exercise 2.36: Give an implementation of the *lookup* procedure for the case where the set of records is structured as a binary tree, ordered by the numerical values of the keys.

2.2.6. Example: Huffman Encoding Trees

This section provides practice using list structure and data abstraction to manipulate sets and trees. The application is to methods for representing data as sequences of 1's and 0's (bits). For example, the ASCII standard code used to represent text in computers encodes each character as a sequence of 7 bits. Using 7 bits allows us to distinguish 2^7, or 128 possible different characters. In general, if we want to distinguish N different symbols, then we will need to use $\log_2 N$ bits per symbol. For example, if all our messages are made up of the 8 symbols A, B, C, D, E, F, G, H, then we can choose a code with 3 bits per character, for example:

A	000	C	010	E	100	G	110
B	001	D	011	F	101	H	111

Using this code, the message

BACADAEAFABBAAAGAH

is encoded as the string of 54 bits

001000010000011000100000101000001001000000000110000111

Codes such as ASCII and the A through H code above are known as *fixed-length* codes, which is to say, they represent each symbol in the message using the same number of bits. It is sometimes advantageous to use *variable-length* codes, in which different symbols may be represented by different numbers of bits. For example, Morse code does not use the same number of dots and dashes for each letter of the alphabet. In particular, E, the most frequent letter, is represented by a single dot. In general, if our messages are such that some symbols appear very frequently and some symbols appear very rarely, then we can encode data more efficiently (i.e., using fewer bits per message) if we assign shorter codes to the frequent symbols.

Consider the following alternative code for the letters A through H:

A	0	C	1010	E	1100	G	1110
B	100	D	1011	F	1101	H	1111

With this code, the same message as above is encoded as the string

100010100101101100011010100100000111001111

which contains 42 bits, a savings of more than 20% in space over the fixed-length code shown above.

One of the problems in using a variable-length code is knowing when you have reached the end of a symbol in reading a sequence of 0's and 1's. Morse code solves this problem by using a special *separator code* (in this case, a pause) between the sequence of dots and dashes for each letter. Another way to solve the problem is to design the code in such a way that no complete code for any symbol is the beginning (or *prefix*) of the code for another symbol. Such a code is called a *prefix code*. For instance, in the example above, A is encoded by 0, so no other symbol can have a code that begins with 0.

In general, we can attain significant savings if we use variable-length prefix codes that take advantage of the relative frequencies of the symbols in the messages to be encoded. One particular scheme for doing this is called the *Huffman Encoding Method*, after its discoverer, David Huffman.

A Huffman code can be represented as a binary tree whose leaves are the symbols that are encoded. At each node of the tree there is a set which is the union of all the symbols in the leaves that lie below the node. In addition, each symbol at a leaf is assigned a *frequency number*, and each node contains a weight that is the sum of all the frequencies of the leaves lying below it. The weights are not used in the encoding or decoding process. We will see below how they are used to help construct the tree.

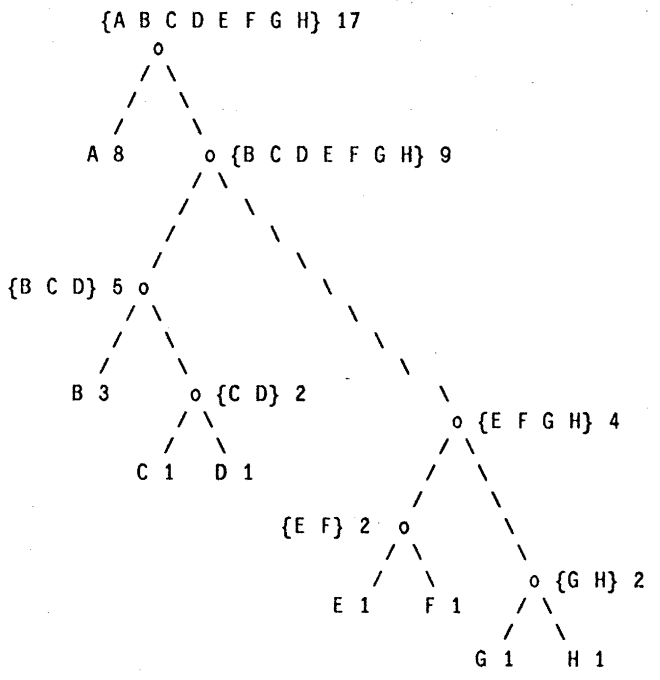


Figure 2-9: A Huffman Encoding Tree

Figure 2-9 shows the Huffman tree for the A through H code given above. The frequency numbers at the leaves indicate that the tree was designed for messages in which A appears with relative frequency 8, B appears with relative frequency 3, and the other letters each appear with relative frequency 1.

Given a Huffman tree, we can find the encoding of any symbol by starting at the root of the tree and following down until we reach the leaf that holds the symbol. Each time we move down a left branch we add a 0 to the code, and each time we move down a right branch we add a 1 to the code. (We can decide which branch to follow by testing to see if the symbol is contained in the sets specified for the branches.) For example, starting from the root of the tree in figure 2-9 we arrive at the leaf for D by following a right branch, then a left branch, then a right branch, then a right branch. Hence the code for D is 1011.

To decode a bit sequence using a Huffman tree, we begin at the root of the tree, and use the successive 0's and 1's of bit sequence to determine whether to move down the left or right branch. Each time we come to a leaf, we have generated a new symbol in the message, at which point we start over from the root of the tree to find the next symbol. For example, suppose we are given the tree above, and the sequence 10001010. Starting at the root, we move down the right branch, since the first bit of the string is 1, then down the left branch, since the second bit is 0, then down the left branch, since the third bit is also 0. This brings us to the leaf for B, so the first symbol of the decoded message is B. Now we start again at the root, and make a left move, since the next bit in the string is 0. This brings us to the leaf for A. Then we start again at the root with the rest of the string 1010, so we move right, left, right, left and reach C. So the entire message is BAC.

Generating Huffman trees

Now that we've seen how to use Huffman trees to encode and decode messages, the question remains: given an "alphabet" of symbols and their relative frequencies, how do we construct the "best" code? In other words, which tree will encode messages using the fewest number of bits? Huffman gave an algorithm for doing this, and showed that the resulting code is indeed the best variable-length code for messages where the relative frequency of the symbols matches the frequencies with which the code was constructed. We will not prove this optimality of Huffman codes here, but we will show how Huffman trees are constructed.[17]

The algorithm for generating a Huffman tree is in fact very simple. The idea is to arrange the tree so that the symbols with the smallest frequency appear farthest away from the root. We can do this as follows: Begin with the set of leaf nodes, together with their frequencies, as determined by the initial data from which the code is to be constructed. Now find two leaves with the smallest frequencies and "merge" them to produce a node that has these two nodes as its left and right branches. The "weight" of the new node is the sum of the two frequencies. Remove the two leaves from the original set, and replace them by this new node. Now continue this process. At each step merge two nodes with the smallest weights, removing them from the set, and replacing them by a node which has these two as its left and right branches. The process stops when there is only one node left, which is the root of the entire tree.

For example, here is how the Huffman tree of figure 2-9 was generated:

```
Initial leaves      <(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)>
Merge               <(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)>
Merge               <(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)>
Merge               <(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)>
Merge               <(A 8) (B 3) ({C D} 2) ({E F G H} 4)>
Merge               <(A 8) ({B C D} 5) ({E F G H} 4)>
Merge               <(A 8) ({B C D E F G H} 9)>
Final merge         <({A B C D E F G H} 17)>
```

Notice that the algorithm does not result in a unique tree because there may not be unique smallest weight nodes at each step. Also, the choice of the order in which the two nodes are merged (which will be the right branch and which will be the left branch is arbitrary.

Representing Huffman trees

For the following exercises we will work with a system that uses Huffman trees to encode and decode messages, and which generates Huffman trees according to the algorithm outlined above. We'll begin by discussing how trees are represented.

Leaves of the tree are represented by a list consisting of the symbol *leaf*, the symbol at the leaf, and the weight:

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (leaf? object)
  (and (not (atom? object))
       (eq? (car object) 'leaf)))
```

---

[17] See the book by Hamming [16] for a discussion of the mathematical properties of Huffman codes.

```
(define (symbol-leaf x) (cadr x))

(define (weight-leaf x) (caddr x))
```

A general tree will be a list of a left branch, a right branch, a set of symbols, and a weight. The set of symbols will be simply a list of the symbols (rather than some more sophisticated set representation). Observe that when we make a tree by merging two nodes, we obtain the weight of the tree as the sum of the weights of the nodes, and the set of symbols as the union of the symbols for the nodes. Since our symbol sets are represented as lists, we can form the union by using the *append* procedure we defined in section 2.2.1.

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))
```

If we make a tree in this way, we have the following selectors:

```
(define (left-branch tree) (car tree))

(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))

(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

Observe that *symbols* and *weight* must do something slightly different depending on whether they are called with a leaf or a general tree. These procedures are simple examples of *generic operators*, about which we will have much more to say in section 2.3.

The decoding procedure

The following procedure implements the decoding algorithm specified above. It takes as arguments a list of 0's and 1's, together with a Huffman tree:

```
(define (decode bits tree)
  (decode-1 bits tree tree))
```

The procedure *decode-1* takes three arguments: the list of bits, the tree, and the current position in the tree. It keeps moving "down" the tree, choosing a left or right branch, according to whether the next bit in the list is a 0 or a 1. (This is done using the subprocedure *choose-branch*.) When it reaches a leaf, it returns the symbol at that leaf as the next symbol in the message (*cons*ing it onto the rest of the message) and proceeds to decode the rest of the message, starting at the root of the tree.

```
(define (decode-1 bits tree current-branch)
  (if (null? bits)
      '()
      (let ((next-branch
             (choose-branch (car bits) current-branch)))
        (if (leaf? next-branch)
            (cons (symbol-leaf next-branch)
                  (decode-1 (cdr bits) tree tree))
            (decode-1 (cdr bits) tree next-branch)))))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit -- CHOOSE-BRANCH" bit))))
```

Note the error check in the final clause of *choose-branch*, which complains if the procedure finds something other than a 0 or a 1 as input data.

Sets of weighted elements

In our representation of trees, each node contains a set of symbols, which we have represented as a simple list. However, the tree-generating algorithm discussed above requires that we also work with sets of leaves and trees, successively merging the two smallest items. Since we will be required to repeatedly find the smallest item in a set, it is convenient to use an *ordered* representation for this kind of set.

We'll represent a set of leaves and trees as a list of elements, arranged in increasing order of weight. The following *adjoin-set* procedure for constructing sets is similar to the one described in exercise 2-33, except that items are compared by comparing their *weights* and the element being added to the set is never already in it:

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
```

The following procedure takes as its argument a list of symbol-frequency pairs, such as

```
((A 4) (B 2) (C 1) (D 1))
```

and constructs an initial ordered set of leaves, ready to be merged according to the Huffman algorithm:

```
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair)     ;symbol
                               (cadr pair))   ;frequency
                    (make-leaf-set (cdr pairs))))))
```

Exercise 2-37: Define an encoding tree and a sample message:

```
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                  (make-code-tree (make-leaf 'B 2)
                                  (make-code-tree
                                   (make-leaf 'D 1)
                                   (make-leaf 'C 1)))))

(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
```

Use the *decode* procedure to decode the message, and give the result.

Exercise 2-38: Write an *encode* program that takes as argument a message and a tree and produces the list of bits that gives the encoded message. The top level of *encode* is as follows:

```
(define (encode message tree)
  (if (null? message)
      nil
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree))))
```

Encode-symbol is a procedure, which you must write, that returns the list of bits that encodes a given symbol according to a given tree. You should design *encode-symbol* so that it signals an error if the symbol is not in the tree at all. Test your procedure by encoding the result you obtained in exercise 2-37 with the sample tree and seeing whether it is the same as the original sample message.

Exercise 2-39: Write a procedure that takes as argument a list of symbol-frequency pairs, and generates a Huffman encoding tree, according to the Huffman algorithm. (Assume that no symbol appears in more than one pair.) The top level procedure is

```
(define (generate-huffman-tree pairs)
  (car (successive-merge (make-leaf-set pairs))))
```

where *make-leaf-set* is the procedure that transforms the list of pairs into an ordered set of leaves as described above. *Successive-merge* is the procedure that you must write, using *make-code-tree* to successively merge the smallest weight elements of the set until there is only one element left, which is the desired Huffman tree.

Note: This procedure is slightly tricky, but not really complicated. If you find yourself designing a complex procedure, then you are almost certainly doing something wrong. You should realize that you can make significant advantage of the fact that we are using an *ordered* set representation.

Exercise 2-40: The following 8-symbol alphabet with associated relative frequencies was designed to efficiently encode the lyrics of 1950's rock songs. (Note that the "symbols" of an "alphabet" need not be individual letters.)

A	2	NA	16
BOOM	1	SHA	3
GET	2	YIP	10
JOB	2	WAH	1

Generate a corresponding Huffman tree, and use it to encode the following message:

Get a job
Sha na na na na na na na na
Get a job
Sha na na na na na na na na
Wah yip yip yip yip yip yip yip yip yip
Sha boom

How many bits are required for the encoding? What is the smallest number of bits that would be needed to encode this song if we used a fixed-length code for the 8-symbol alphabet?

Exercise 2-41: Suppose we have a Huffman tree for an alphabet of N symbols, and that the relative frequencies of the symbols are 1, 2, 4, ..., 2^{N-1}. Sketch the tree for N = 5; for N = 10. In such a tree (for general N) how may bits are required to encode the most frequent symbol? the least frequent symbol?

Exercise 2-42: Consider the encoding procedure that you designed in exercise 2-38. What is the order of growth in the number of steps needed to encode a symbol? Be sure to include the number of steps needed to *search* the symbol list at each node encountered. To answer this question in general is difficult. Consider the special case where the relative frequencies of the N symbols are as described in exercise 2-41, and give the order of growth (as a function of N) of the number of steps needed to encode the most frequent and least frequent symbols in the alphabet.

2.3. 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 independently of the choices involved in implementing the data objects that the program manipulates. For example, we saw in section 2.1.1 how to attack the problem of designing a program that uses rational numbers, separately from the problem of implementing rational numbers in terms of the computer language's primitive mechanisms for constructing compound data. The key idea is to erect an *abstraction barrier*, in this case, the selectors and constructors for rational numbers (`make-rat`, `numer`, `denom`), that isolates the way that rational numbers are used from their underlying representation in terms of list structure. A similar abstraction barrier isolates the details of the procedures that perform rational arithmetic (`+rat`, `-rat`, `*rat`, and `/rat`) from the "higher level" procedures that use rational numbers. As we saw, the resulting program has the structure shown in figure 2-1.

These data abstraction barriers are powerful tools for controlling complexity. By isolating the underlying representations of data objects, we can divide the problem of designing a large program into smaller problems that can be solved separately. But the kind of data abstraction we have presented is not yet powerful enough. In a large system it may not make sense to speak of "the underlying representation" of a data object. 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 operators for manipulating complex numbers will work with *either* representation.

Now 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 operators* -- that is, procedures that can operate on data that may be represented in more than one way. Our main technique for building generic operators will be to work in terms of data objects that have *manifest types*, 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 systems of generic operators.

We begin our discussion with the simple complex number example. We will see how manifest types and data-directed style enable us to design separate rectangular and polar representations for complex numbers, while still maintaining the notion of an abstract "complex number" data object. We will accomplish this by defining arithmetic operators for complex numbers `+c`, `-c`, `*c`, and `/c`, in terms of generic selectors that access parts of a complex number independently of how the number is represented. The resulting complex number system, as shown in figure 2-10, contains two different kinds of abstraction barriers. The "horizontal" abstraction barriers play the same role as the ones in in figure 2-1. They

isolate "higher level" operations from "lower level" representations. In addition, there is a "vertical" barrier that gives us the ability to separately design and install alternative representations.

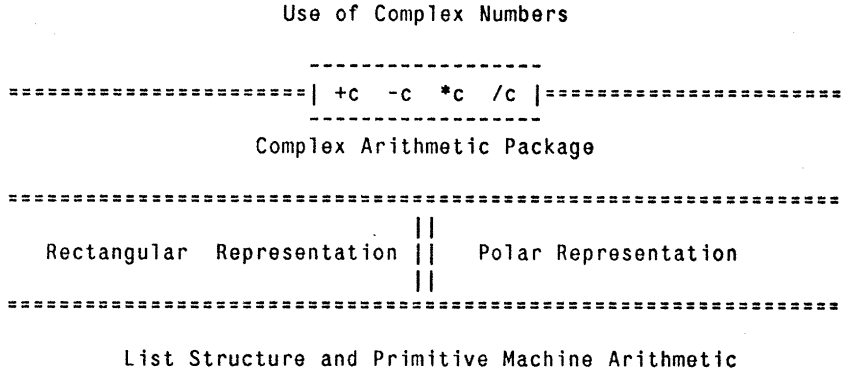


Figure 2-10: Data abstraction barriers in complex number system

In section 2.4 we will show how to use manifest types and data-directed style to develop a generic arithmetic package. This provides operators *add*, *mul*, and so on, which can be used to manipulate all sorts of "numbers," and which can be easily extended when a new kind of number is needed. Figure 2-11 shows the structure of the system we shall build. Notice the abstraction barriers: from the perspective of someone using "numbers," there is a single operator *add* that operates on whatever numbers are supplied. In fact, *add* is a "generic interface" that allows the separate real, rational, and complex arithmetic packages to be accessed uniformly by programs that use numbers. Moreover, any individual arithmetic package (such as the complex package) may itself be accessed through generic operators (such as +c) that combine packages designed for different representations. Of particular importance to the system designer is the fact that one can design the individual arithmetic packages separately, and combine them to produce a generic arithmetic package by using data-directed style as a conventional interface.

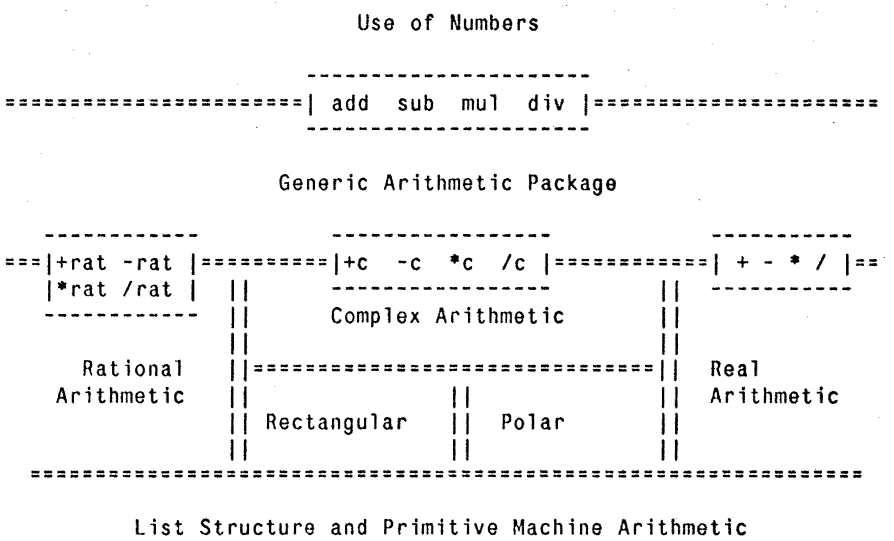


Figure 2-11: Generic arithmetic system

2.3.1. Representations for Complex Numbers

We will develop a system that performs arithmetic operations on complex numbers as a simple, albeit somewhat unrealistic, example of a program that uses generic operators. 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).[18] In section 2.3.2 we show how both both representations can be made to coexist in a single system, through the use of manifest types and generic operators. In section 2.3.3 , we introduce data-directed programming style as an technique for organizing systems that use generic operators.

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 2-12.) From this point of view, the complex number

$$z = x + i\,y \quad \text{where } i^2 = -1$$

can be thought of as the vector whose real coordinate is x and whose imaginary coordinate is y. Addition of complex numbers reduces in this representation to addition of coordinates. That is,

$$\text{Real-part}(z_1+z_2) = \text{Real-part}(z_1) + \text{Real-part}(z_2)$$

$$\text{Imaginary-part}(z_1+z_2) = \text{Imaginary-part}(z_1) + \text{Imaginary-part}(z_2)$$

---

[18] In actual computational systems, rectangular form is preferable to polar form most of the time because of round-off 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 operators, as well as a good introduction to the more substantial systems developed later in this chapter.

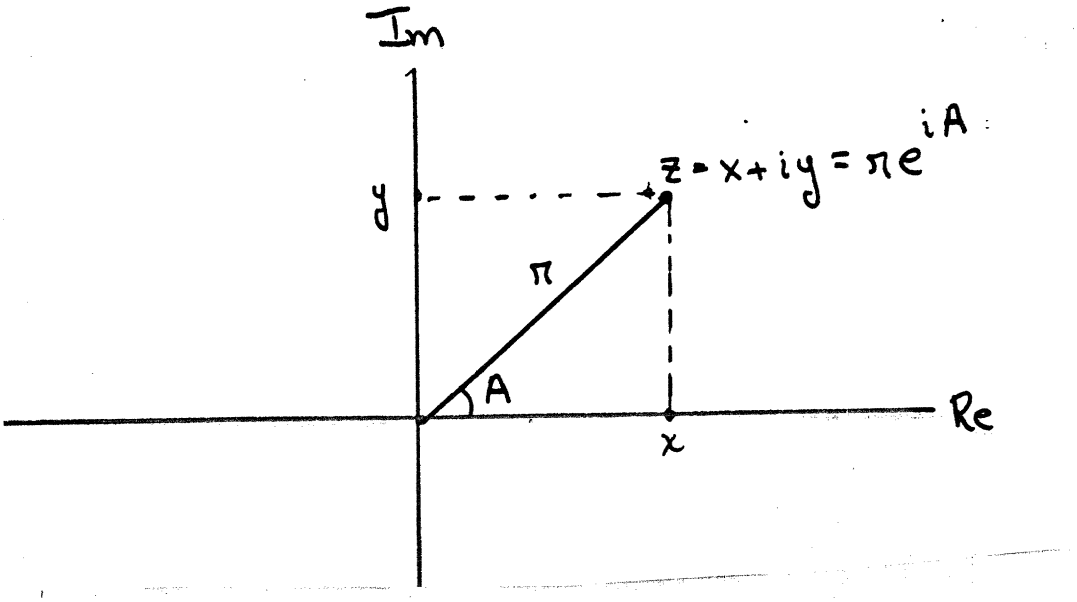


Figure 2-12: Complex numbers as vectors

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 as shown in figure 2-12. 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. That is,

$$\text{Magnitude}(z_1 * z_2) = \text{Magnitude}(z_1) * \text{Magnitude}(z_2)$$

$$\text{Angle}(z_1 * z_2) = \text{Angle}(z_1) + \text{Angle}(z_2)$$

We see therefore that there are two different representations for complex numbers, which are appropriate for different operations. And yet, from the point of view of someone writing a program that uses complex numbers, the principle of data abstraction suggests that the complex number manipulation procedures be ambiguous with respect to the actual implementation used by the computer. For example, it is often useful to be able to find the magnitude of a complex number that is specified by rectangular coordinates. Similarly, it is often useful to be able to determine the real part of a complex number that is specified by polar coordinates.

To design such a system, we can follow the same data abstraction strategy as we did in designing the rational number package in section 2.1.1. We assume that the operators on complex numbers are implemented in terms of the following four selectors: *real-part*, *imag-part*, *magnitude*, and *angle*. We'll also assume that we have two procedures for constructing complex numbers. The first, *make-rectangular*, returns a complex number with given real and imaginary parts, and the second, *make-polar*, returns a complex number with given magnitude and angle. We have, for any complex number *z*:

```
(make-rectangular (real-part z) (imag-part z)) => z
(make-polar (magnitude z) (angle z))           => z
```

Using these constructors and selectors, we can implement complex number arithmetic using the "abstract data" specified by the constructors and selectors, just as we did for rational numbers in section 2.1.1. As shown in the formulas above, we can add and subtract complex numbers in terms of real and imaginary parts, while multiplying and dividing complex numbers in terms of magnitudes and angles:

```
(define (+c z1 z2)
  (make-rectangular (+ (real-part z1) (real-part z2))
                    (+ (imag-part z1) (imag-part z2))))

(define (-c z1 z2)
  (make-rectangular (- (real-part z1) (real-part z2))
                    (- (imag-part z1) (imag-part z2))))

(define (*c z1 z2)
  (make-polar (* (magnitude z1) (magnitude z2))
              (+ (angle z1) (angle z2))))

(define (/c z1 z2)
  (make-polar (/ (magnitude z1) (magnitude z2))
              (- (angle z1) (angle z2))))
```

To complete the complex arithmetic package, we must choose a representation, and implement the constructors and selectors in terms of the primitive numbers and list structure. There are two obvious possible choices. 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? Just as in our discussion of rational numbers, we can decide about the implementation independently of our decisions about how we support the abstract selectors and constructors for that representation.

Let's consider the consequences of the various choices. If we represent a complex number in rectangular form, then selecting the real and imaginary parts is straightforward, as is constructing a complex number with a given real and imaginary part. To find the magnitude and angle, or to construct a complex number with a given magnitude and angle, we use the 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 angle (r, A). This leads to the following selectors:

```
(define (make-rectangular x y) (cons x y))

(define (real-part z) (car z))

(define (imag-part z) (cdr z))

(define (make-polar r a)
  (cons (* r (cos a)) (* r (sin a))))

(define (magnitude z)
  (sqrt (+ (square (car z)) (square (cdr z)))))
```

```
(define (angle z)
  (atan (cdr z) (car z)))
```

On the other hand, we may choose to implement our complex numbers in polar form. If so, then selecting the magnitude and angle will be straightforward, while we must use trigonometry to find the real and imaginary parts. Here is the corresponding set of procedures:

```
(define (make-rectangular x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))

(define (real-part z)
  (* (car z) (cos (cdr z))))

(define (imag-part z)
  (* (car z) (sin (cdr z))))

(define (make-polar r a) (cons r a))

(define (magnitude z) (car z))

(define (angle z) (cdr z))
```

As usual, the discipline of data abstraction ensures that the implementation of the complex number arithmetic operators `+c`, `-c`, `*c`, and `/c` is independent of which representation we choose.

2.3.2. Manifest Types

One way to view data abstraction is as an application to program design of the "principle of least commitment." By setting up selectors and constructors as an abstraction barrier we can 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. In fact, the principle of least commitment can be carried to further extremes than we have seen so far. If we desire, we can maintain the ambiguity of representation even *after* we have designed the selectors and constructors, electing to represent some complex numbers in polar form and some in rectangular form. But if both kinds of representations are included in a single system, we will need some way to distinguish data constructed by *make-polar* from data constructed by *make-rectangular*. Otherwise, if we were asked, for instance, to find the *magnitude* of *(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" -- *rectangular* or *polar* -- as part of each complex number. Then, when we need to manipulate a complex number, we can use the type to decide which selector to apply.

A data object that has a type that can be recognized and tested is said to have *manifest type*. In order to manipulate typed data, we will assume that we have two procedures, *type* and *contents*, that extract from a datum the type and the actual contents (the list of

coordinates in the case of a complex number). We will also postulate a procedure *attach-type* that takes a type and a contents and produces a typed datum. A straightforward way to implement this is to use ordinary list structure for this purpose:

```
(define (attach-type type contents)
  (cons type contents))

(define (type datum)
  (if (not (atom? datum))
      (car datum)
      (error "Bad typed datum -- TYPE" datum)))

(define (contents datum)
  (if (not (atom? datum))
      (cdr datum)
      (error "Bad typed datum -- CONTENTS" datum)))
```

Using these procedures, we can define predicates *polar?* and *rectangular?*, which recognize polar and rectangular numbers, respectively.

```
(define (polar? z)
  (eq? (type z) 'polar))

(define (rectangular? z)
  (eq? (type z) 'rectangular))
```

Now we modify the constructors for complex numbers to include the type as part of the number to be constructed. To construct a complex number in rectangular form, given real and imaginary parts, we use:

```
(define (make-rectangular x y)
  (attach-type 'rectangular (cons x y)))
```

To construct a complex number in polar form, given magnitude and angle, we use:[19]

```
(define (make-polar r a)
  (attach-type 'polar (cons r a)))
```

Now we can use the type of a complex number to select the specific procedures for dealing with numbers of the given type. We use *contents* to get at the bare, untyped datum. Our abstract selectors for complex numbers are now defined in terms of the appropriate selectors for the untyped complex numbers. We see that these procedures can be divided into two "packages," one for handling rectangular form and the other for polar form.

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))))
```

[19] In our previous implementation, we also included operations for constructing a rectangular number from a magnitude and angle, and for constructing a polar number from real and imaginary parts. These will be unnecessary in the new system we are designing.

```
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))))
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))))
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))))
```

As the underlying procedures in each of the packages for handling untyped complex numbers, we can use the selectors defined in the previous section, after renaming each procedure so as to avoid name conflicts. Here are the selectors for the rectangular representation:

```
(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (car z))
           (square (cdr z)))))

(define (angle-rectangular z)
  (atan (cdr z) (car z)))
```

The selectors for the polar representation are as follows:

```
(define (real-part-polar z)
  (* (car z) (cos (cdr z))))

(define (imag-part-polar z)
  (* (car z) (sin (cdr z))))

(define (magnitude-polar z) (car z))

(define (angle-polar z) (cdr z))
```

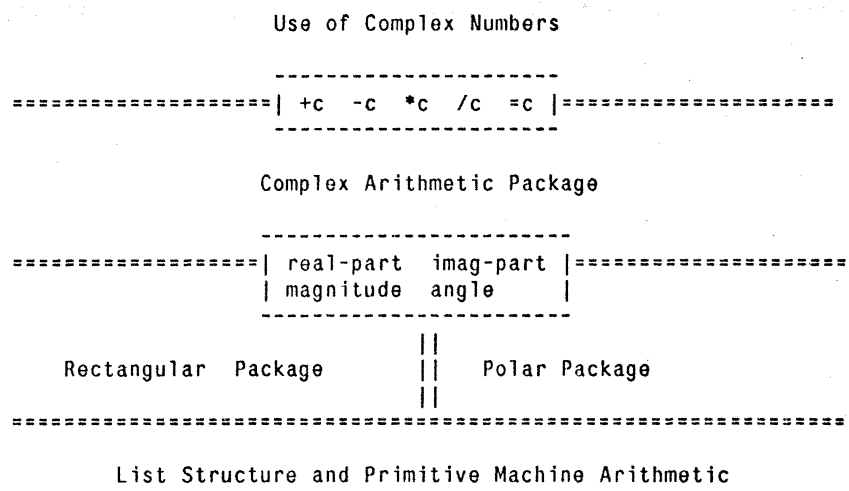


Figure 2-13: Structure of the generic complex arithmetic system.

The resulting complex number system has the structure shown in figure 2-13. Notice that the system can be decomposed into three relatively independent parts: the complex number arithmetic package, the rectangular representation package, and the polar representation package. Each of these packages could have been designed without any knowledge of the others. For instance, the polar and rectangular packages could have been written separately, by two separate people, and then *both* could be used as underlying representations by a third programmer who implements the complex arithmetic procedures *+c*, *-c*, **c*, and */c*, in terms of the abstract constructor/selector interface.

Since each data object is "tagged" with its type, the abstract selectors can operate on the data in a generic manner. That is to say, each selector may be 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 packages: Within a given representation package (say, the 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 type and passes the untyped contents on to the polar package. Conversely, when a number is constructed and "exported" from the polar package, it is given a manifest type so that it can be appropriately recognized by the higher level procedures. This discipline of stripping off and attaching types as data is passed from level to level will be an important organizational strategy, as we shall see in section 2.4 below.

Although this way of organizing generic operators is very valuable, there are two weak points in our system. One is that the generic interface procedures (*real-part*, *imag-part*, *magnitude*, *angle*) must "know about" all the different representations. In the following section, we will introduce data-directed programming, a technique which can be used to deal with this problem. Another weakness of our system is that although the separate packages can be designed separately, we have to make sure that no two procedures in the entire system have the same name. This is why we appended the package name to each selector

procedure in the example above.[20]

2.3.3. Data-directed Programming

Using manifest types and generic operators is a powerful tool for obtaining modularity in system design. But the techniques we have available at this moment are too weak to solve really large-scale problems. For example, suppose someone designed a new package, using a new representation for complex numbers, and asked us to interface this to our complex number system. We would need to identify this new representation with a type, and then add a clause to each of the generic interface procedures (*real-part*, *imag-part*, *magnitude*, *angle*) to check for the new type and access the appropriate selector in the new package. This is not much of a problem for the complex number system as it stands, but suppose that 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 knows all the interface procedures nor all the representations. Although this is not likely to be the case with systems that perform arithmetic, the problem is real and must be addressed in programs such as large-scale data base management systems and symbolic algebra systems. What we need is a means for modularizing the system design even further. This is what is provided by the programming technique known as *data-directed programming*.

To understand how data-directed programming works, we begin with the observation that, whenever we deal with a number of generic operators that are common to a number of different types, we are, in effect, dealing with a two-dimensional table that contains the possible operators on one axis and the possible types on the other axis. The entries in the table are the procedures that implement each operator for each type of operand presented. In the complex number system developed in the previous section, the correspondence between operator name, data type, and actual procedure was spread out among the various conditional clauses in the generic interface procedures. But the same information could have been organized in a table, as shown in figure 2-14.

		types	
		polar	rectangular
operators	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 2-14: Table of Operators for Complex Number System

Data-directed programming is the technique of designing our programs to work with such a table directly. Previously, we implemented the mechanism that interfaces the complex

---

[20] There are more elegant ways to handle the name conflict problem. In Chapter 3, we will see how *environments* serve as contexts that determine the meaning of names in expressions to be evaluated. We can take advantage of this idea to show how to design each package to be a separate environment with its own local names, and how operators from different packages can be combined.

arithmetic package to the two representation packages as a set of procedures. Instead, we will implement the interface package by a single procedure that looks up the operator/type combination in the operator table to find the correct procedure to apply, and then applies it to the contents of the operand. If we do this, then in order to add a new representation package to the system, we need not change any existing procedures, but only add new entries to the table.

To implement this plan, we'll assume that we have two procedures *put* and *get* for manipulating the operator/type table:

(put <type> <op> <item>)
: installs *<item>* in the table entry indexed by *<type>* and *<op>*

(get <type> <op>)
: looks up the *<type>*, *<op>* entry in the table and returns the item found there. If no item is found, *get* returns *nil*.

For now, we can assume that *put* and *get* are primitive operators 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 the data-directed system works: the programmer who defined the rectangular representation package could install it in the complex arithmetic system by adding entries to the table that tells the system how to operate on rectangular numbers:

```
(put 'rectangular 'real-part real-part-rectangular)
(put 'rectangular 'imag-part imag-part-rectangular)
(put 'rectangular 'magnitude magnitude-rectangular)
(put 'rectangular 'angle angle-rectangular)
```

Notice that the *<item>* entries in the table are the actual procedures that are to be applied.

Meanwhile, another programmer could work on the polar form definitions, independently of his colleague, and the completed definitions could be similarly interfaced to the complex number package:

```
(put 'polar 'real-part real-part-polar)
(put 'polar 'imag-part imag-part-polar)
(put 'polar 'magnitude magnitude-polar)
(put 'polar 'angle angle-polar)
```

The complex arithmetic package itself accesses the table by means of a general "operator" procedure called *operate*. *Operate* "applies a generic operator to an object" by looking in the table under the name of the operator and the type of the object, and applying the resulting procedure if one is present:

```
(define (operate op obj)
  (let ((proc (get (type obj) op)))
    (if (not (null? proc))
        (proc (contents obj))
        (error "Operator undefined for this type -- OPERATE"
               (list op obj)))))
```

Using *operate*, our generic interface procedures are defined as follows:

```
(define (real-part obj) (operate 'real-part obj))
(define (imag-part obj) (operate 'imag-part obj))
(define (magnitude obj) (operate 'magnitude obj))
(define (angle obj) (operate 'angle obj))
```

In particular, these procedures do not have to be changed at all if a new representation is added to the system.

The general strategy of checking the type of a datum and calling an appropriate procedure, is called "dispatching on type," and data-directed programming is an extremely flexible way to organize the dispatch. This kind of "conventional interface" can be used to combine packages for representations that were separately constructed. This technique is used regularly by expert programmers to enhance the extensibility and modularity of their systems.

Exercise 2-43: In section 2.2.4 we described a program that performs symbolic differentiation:

```
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))

        <more rules can be added here>
        ))
```

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

```
(define (deriv exp var)
   (cond ((constant? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var) 1 0))
         (else ((get (operator exp) 'deriv) (operands exp) var))))

(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

a. Explain what was done above. Why can't we assimilate the *constant?* and *same-var?* predicates into the data-directed dispatch?

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

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

d. In this simple algebraic manipulator the "type" of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the procedures in the opposite way, where the dispatch line looked like:

```
((get 'deriv (operator exp)) (operands exp) var)
```

What are the corresponding required changes to the rule definitions?

Exercise 2-44: 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, was dismayed to discover that while all the division files were implemented as data structures in Lisp, the particular data structure used varied from division to division. A meeting of division managers was hastily called, to search for a strategy to integrate the files, that would satisfy headquarter's needs while preserving the existing autonomy of the divisions.

Show how such a strategy can be implemented using data-directed programming. As an example, suppose that each division's personnel records are contained in a single personnel file, which contains a set of records, keyed on employee names, where the structure of the set varies from division to division. Furthermore, each employee record is itself a set (structured differently from division to division) that contains information keyed under identifiers such as `address`, `salary`, and so on.

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

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

c. Implement for headquarters a `find-employee-record` procedure that searches *all* the division files for the record of a given employee and returns the record. Assume that this procedure takes as arguments an employee name, and a list of all the division files.

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

Message Passing

The key idea of data-directed programming is that we handle generic operators in our programs by dealing explicitly with operator/type tables, such as the table in figure 2-14. The more traditional style of programming, which we used in section 2.3.2, organized the required dispatching on type by having each operator take care of its own dispatching. In effect, this style of programming decomposes the operator/type table into rows, with each generic operator procedure representing a row of the table.

An alternative implementation strategy is to decompose the table into columns: Instead of using "intelligent operators" that dispatch on data types, we work with "intelligent data objects" that dispatch on operator names. We can do this by arranging things so that a data object, such as a rectangular number, is represented as a *procedure* that takes as input the required operation name, and performs the operation indicated. In such a discipline, `make-rectangular` could be written as

```
(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "Unknown op -- MAKE-RECTANGULAR" m))))
  dispatch)
```

The corresponding *operate* procedure, which applies a generic operation to a data object, now simply feeds the operation name to the data object and lets the object do the work:

```
(define (operate op obj) (obj op))
```

Note that the "data object" returned by *make-rectangular* is a *procedure* -- the internal *dispatch* procedure. This is the procedure that is invoked when *operate* requests an operation to be performed.

This style of programming is called *message passing*. The name comes from the image that a data object is an entity that receives the requested operation name as a "message." We have already seen an example of message passing, in section 2.1.3, where we used it to show how *cons*, *car*, and *cdr* could be defined without using any data objects, but only procedures. Here we see that message passing is not simply a mathematical trick, but a useful technique for organizing systems that must cope with generic operators. In the remainder of this chapter, we will continue to use data-directed programming, rather than message passing, to discuss generic arithmetic operators. In Chapter 3 we will return to message passing, and see that it can be a powerful tool for structuring simulation programs.

Exercise 2-45: Implement the constructor *make-polar* in message passing style, analogous to the *make-rectangular* procedure given above.

Exercise 2-46: As a large system with generic operators evolves, we may need to add new types of data objects, or add new operators. For each of the three organizational strategies -- "conventional" style (as in section 2.3.2), 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 operators. Which organization would be most appropriate for a system in which we must often add new types? Which would be most appropriate for a system in which we must often add new operators?

2.4. Systems with Generic Operators

In the previous section, we saw how to design systems in which data objects can be represented in more than one way. The key idea is to link the package that specifies the data operations to the several packages that implement the various representations by means of generic interface procedures. We'll now see how to use this same idea, not only to define operators that are generic over different representations, but also to define operators which are generic over different kinds of operands.

We'll consider the problem of designing a set of arithmetic operators that work on "all different kinds of numbers." We have already seen several different packages of arithmetic

operators. There is the primitive arithmetic (+, -, *, /) built into our language, the rational number arithmetic (*+rat*, *-rat*, **rat*, */rat*) that we implemented in section 2.1.1, and the generic complex number arithmetic that we implemented in the previous section. We will now use data-directed techniques to construct a package of arithmetic operators that incorporates all of the arithmetic systems we have already constructed. Moreover, our operators will be "extensible" in the sense that if later we come up with a new class of "numbers" we can easily add these to the system without changing any of the programs we have already written.

2.4.1. Generic Arithmetic Operators

The problem of designing generic arithmetic operators is analogous to the problem of designing the generic complex number operators. We would like, for instance, to have a generic addition operator *add*, which acts like ordinary primitive addition + on ordinary numbers, like *+rat* on rational numbers, and like *+c* on complex numbers. We can implement *add*, and the other generic arithmetic operators, by following the same strategy that we used in the previous section to implement the generic selectors for complex numbers. We'll attach a manifest type to each kind of number, and cause the generic operator to dispatch to an appropriate package, according to the data type of its arguments.

Let's begin by installing a package for handling "ordinary numbers," that is, the primitive numbers of our language. We'll refer to these as type *number*. The arithmetic operators in this package are essentially the primitive arithmetic:

```
(define (+number x y)
   (make-number (+ x y)))

(define (-number x y)
   (make-number (- x y)))

(define (*number x y)
   (make-number (* x y)))

(define (/number x y)
   (make-number (/ x y)))
```

Here *make-number* is a procedure that attaches an appropriate manifest type to its argument:

```
(define (make-number n)
   (attach-type 'number n))
```

The next step is to link the operators in the package to the generic operators *add*, *sub*, *mul*, and *div*. We do this with data-directed programming, just as in section 2.3.3. As before, we place the procedures in a table, indexed under the data type and generic operator name:

```
(put 'number 'add +number)
(put 'number 'sub -number)
(put 'number 'mul *number)
(put 'number 'div /number)
```

The actual generic operators are defined as follows:

```
(define (add x y) (operate-2 'add x y))
(define (sub x y) (operate-2 'sub x y))
(define (mul x y) (operate-2 'mul x y))
(define (div x y) (operate-2 'div x y))
```

As with the complex number selectors, our generic arithmetic operators will use a general "operate" procedure that dispatches according to the type of the argument. However, while the selectors for complex numbers were operators with one argument, our generic arithmetic operators are operators with two arguments. Hence we cannot use the same *operate* procedure as before (section 2.3.3). Instead, we use the following procedure to perform the dispatch:

```
(define (operate-2 op arg1 arg2)
  (let ((t1 (type arg1)))
    (if (eq? t1 (type arg2))
        (let ((proc (get t1 op)))
          (if (not (null? proc))
              (proc (contents arg1) (contents arg2))
              (error
               "Operator undefined on this type -- OPERATE-2"
               (list op arg1 arg2))))
        (error "Operands not of same type -- OPERATE-2"
               (list op arg1 arg2))))))
```

Operate-2 verifies that the two operands have the same type and, if so, dispatches to the procedure that was installed in the table for the given type and operator. If there is no such procedure then *operate-2* signals an error.

If the two operands do not have the same type, then *operate-2* signals an error. This is not really the correct thing to do. For instance, if we try to add the (primitive) number 3 to the (complex) number 2+4**i**, then *operate-2* will complain that the types do not match. And yet we should expect a "reasonable" system to produce the answer 5+4**i**. On the other hand, it turns out that arranging for this kind of "reasonable" behavior opens an enormous can of worms concerning the interactions among data of different types. We will duck this issue now, and return to it in section 2.4.2.

Exercise 2-47: In defining the package for handling ordinary numbers, we defined operators *+number*, *-number*, and so on, which were essentially nothing more than calls to the primitive operators +, -, etc. It was not possible to use the primitives of the language directly because our manifest type system requires that each data object have a type attached to it. In fact, however, Lisp implementations do have a type system, which they use internally, and Scheme dialect of Lisp includes an operator *primitive-type*, which returns the (internal) type of a data object. For example,

```
==>(primitive-type 3)
number

==>(primitive-type 'apple)
symbol

==>(primitive-type '(a b))
pair
```

Using this operator, modify the definitions of *type*, *contents*, and *attach-type* from section 2.3.2, so that our generic system takes advantage of the internal type system. That is to say, the system should work as before, except that ordinary numbers are represented simply as numbers, rather than as pairs whose *car* is the symbol *number*.

Interfacing the complex number package

Now that the framework of the generic arithmetic system is in place, it is easy to incorporate the complex number package. We begin by writing a procedure that attaches the type *complex* to complex numbers, so they can be recognized outside of the complex package:

```
(define (make-complex z)
  (attach-type 'complex z))
```

We next define the complex number operators to be calls to our generic complex representation operators:

```
(define (+complex z1 z2) (make-complex (+c z1 z2)))
(define (-complex z1 z2) (make-complex (-c z1 z2)))
(define (*complex z1 z2) (make-complex (*c z1 z2)))
(define (/complex z1 z2) (make-complex (/c z1 z2)))
```

Finally, we install the complex arithmetic operators in the appropriate positions in the operator table, so that the generic arithmetic operators will dispatch correctly:

```
(put 'complex 'add +complex)
(put 'complex 'sub -complex)
(put 'complex 'mul *complex)
(put 'complex 'div /complex)
```

What we have here is a two-level type system. A typical complex number is represented in the system as:

```
(complex rectangular 3 4)
```

The outer type (*complex*) is used to direct the number to the complex package. Once within the complex package, the next type (*rectangular*) is used to direct the number to the rectangular package. Strictly speaking, *rectangular* and *polar* are not types of numbers at all, but rather types for the *contents* of a complex number. In a large, complicated system, there might be many levels, each interfaced to the next by means of generic operators. As a data object is passed "downward" the outer type that is used to direct it to the appropriate package is stripped off (by applying *contents*) and the next level of type becomes visible to be used for further dispatching.

Exercise 2-48: When we execute the expression

```
(add (make-complex (make-rectangular 3 4))
     (make-complex (make-polar 5 1)))
```

what are the actual arguments sent to *+c*? What happened to the symbol *complex*? Where was it stripped off?

Finally, observe that the operators *real-part*, *imag-part*, *magnitude*, and *angle* are available only inside the complex number package -- they are defined only for data objects of type *rectangular* or *polar*. On the other hand, it is easy to "export" these operators from the package, so that they can be applied directly to objects of type *complex*, and, in fact, automatically redispatched to the right representation type. We must first define the correct operators:

```
(define (real-part-complex z)
  (make-number (real-part z)))
```

```
(define (imag-part-complex z)
  (make-number (imag-part z)))

(define (magnitude-complex z)
  (make-number (magnitude z)))

(define (angle-complex z)
  (make-number (angle z)))
```

This is accomplished by simply installing these operators in the table under type *complex*:

```
(put 'complex 'real-part real-part-complex)
(put 'complex 'imag-part imag-part-complex)
(put 'complex 'magnitude magnitude-complex)
(put 'complex 'angle angle-complex)
```

Exercise 2-49: Describe in detail why this exporting method works. As an example, trace through all the procedures called in evaluating the expression *(magnitude z)* where *z* is the object *(complex rectangular 3 4)*. In particular, how many time do we go through *operate*? What procedure is dispatched to in each case?

2.4.2. Combining Operands of Different Types

We've seen how to define a unified arithmetic system that encompasses ordinary numbers, complex numbers, rational numbers, and any other type of number we might decide to invent. But we have ignored an important problem. The operators we have defined so far treat the different data types as being completely independent. Thus there are separate packages for adding, say, two ordinary numbers, or two complex numbers. What we have not yet considered is the fact that it is meaningful to define operations that cross the type boundaries, such as the addition of a complex number to an ordinary number. The reason why this is a problem is that we have gone to great pains to introduce barriers between parts of our programs so that they can be developed and understood separately. We would like to introduce the new operations in some carefully controlled way, so that we can support the cross-type operations without seriously violating our module boundaries.

One way to handle cross-type operations is to design a different operator for each possible pair of types for which the operation is valid. For instance, we could have various addition operations *+number-complex* (which adds an ordinary number to a complex number), *+rational-complex*, and so on. Then we can arrange these in a three-dimensional table that indexes the appropriate procedure under the generic operator name, the type of the first argument, and the type of the second argument. Support for such a table is easily introduced by adding a new clause to the *operate-2* procedure of section 2.4.1.

This three-dimensional table method allows us to combine numbers of different types, but at an enormous price. Observe that if there are n different types in our system, then we would need in general to design n^2 different versions of each generic operator. In such a system the cost of introducing a new type is not just the construction of the package of operators for that type but also the construction and installation of the procedures that implement the cross-type operations. This can easily be much more code than is needed to define the operators on the type itself. If our system includes not only binary operators, but also operators on

three, four, or more arguments that may have different types, then the penalty for introducing a new type is even more severe.

Coercion

In the general situation of completely unrelated operations acting on completely unrelated types, the three-dimensional table method for handling operands of different types, cumbersome though it may be, is the best that one can hope for. Fortunately, we can usually do better, by taking advantage of additional structure that may be latent in our type system. Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type for the purposes of performing manipulations. This process is called "coercion." For example, if we are asked to arithmetically combine an ordinary number with a complex number, we can view the ordinary number as a complex number whose imaginary part is 0. This transforms the problem to that of combining two complex numbers, which can be handled in the ordinary way by the complex arithmetic package.

In general, we can implement this idea by designing "coercion" procedures that transform an object of one type to an equivalent object of another type. Here is a typical coercion procedure, which transforms a given ordinary number to a complex number with that real part and zero imaginary part.

```
(define (number->complex n)
  (make-complex (make-rectangular (contents n) 0)))
```

We install these coercion procedures in a special coercion table, indexed under the names of the two types:

```
(put-coercion 'number 'complex number->complex)
```

(We assume that there are procedures *put-coercion* and *get-coercion* available for manipulating this table.) Observe that some of the slots in the table will in general be empty, because it is not possible in general to coerce an arbitrary data object of each type into all other types. For example, there is no way to coerce an arbitrary complex number to an ordinary number, so there will be no general *complex->number* procedure included in the table.

Once the coercion table has been set up, we can handle coercion in a data-directed manner, by modifying the *operate-2* procedure given on page 124 as follows. When asked to operate on two objects *obj1* and *obj2*, we first check to see if they have the same type. If so, we dispatch to the procedure for handling that type, just as before. If the types are different, we check the coercion table to see if objects of type 1 can be coerced to type 2. If so, we coerce *obj1*, and retry the operation. If objects of type 1 cannot in general be coerced to type 2, we try the coercion the other way around, to see if there is a way to coerce *obj2* to the type of *obj1*. Finally, if there is no known way to coerce either type to the other type, we give up. Here is the procedure:

```
(define (operate-2 op obj1 obj2)
  (let ((t1 (type obj1)) (t2 (type obj2)))
    (if (eq? t1 t2)
        (let ((proc (get t1 op)))
          (if (not (null? proc))
              (proc (contents obj1) (contents obj2))
              (error
               "Operator undefined on this type -- OPERATE-2"
               (list op obj1 obj2))))
        (let ((t1->t2 (get-coercion t1 t2))
              (t2->t1 (get-coercion t2 t1)))
          (cond ((not (null? t1->t2))
                 (operate-2 op (t1->t2 obj1) obj2))
                ((not (null? t2->t1))
                 (operate-2 op obj1 (t2->t1 obj2)))
                (else
                 (error "Operands not of same type -- OPERATE-2"
                        (list op obj1 obj2)))))))))
```

This coercion scheme for binary operators has many advantages over the unstructured three-dimensional table method outlined above. Although we still need to write coercion procedures to relate the types (possibly n^2 procedures for a system with n types), we only need to write one procedure for each pair of types, rather than a different procedure for each pair of types and each generic operator.[21] What we are counting on here is the fact that the appropriate transformation between types depends only on the types themselves, not on the operator to be applied.

On the other hand, there may be applications for which our coercion scheme is not general enough. For instance, even when neither of the objects to be combined can be converted to the type of the other, it may still be possible to perform the operation by converting both objects to a third type. In order to deal with such complexity, yet preserve modularity in our programs, it is usually necessary to build systems that take advantage of still further structure in the relations among types, as we discuss next.

Type hierarchies

The coercion scheme presented above relied on the existence of natural relations between pairs of types. Often, there is more "global" structure in how the different types relate to each other. For instance, suppose we are building a generic arithmetic system to handle integers, rational numbers, real numbers, and complex numbers. In such a system, it is quite natural to regard integers as a special kind of rational number, which is in turn a special kind of real number, which is in turn a special kind of complex number. What we actually have is a so-called *hierarchy of types*, in which, for example, integers are a *subtype* of rational

---

[21] In fact, if we are clever, we can usually get by with fewer than n^2 coercion procedures. For instance, if we know how to convert from type 1 to type 2, and from type 2 to type 3, then we can use this to convert from type 1 to type 3. This can greatly decrease the number of conversion procedures we need to supply explicitly when we add a new type to the system. If we are willing to build the required amount of sophistication into our system, we can have it search the "graph" of relations among types, and automatically generate those conversion procedures that can be inferred from the ones that are supplied explicitly.

numbers, meaning that any operator which can be applied to a rational number can automatically be applied to an integer. Conversely, we say that rational numbers form a *supertype* of integers. The particular hierarchy we have here is of a very simple kind, in which each type has at most one supertype and at most one subtype. Such a structure is called a *tower*, and can be illustrated as shown in figure 2-15.

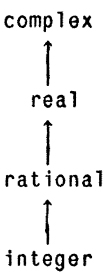


Figure 2-15: A tower of types

If we do have a tower structure, then we can greatly simplify the problem of adding a new type to the hierarchy, for we need only specify how the new type is embedded in the next supertype above it, and how it is the supertype of the type below it. For example, if we want to add an integer to a complex number, we need not explicitly define a special coercion procedure procedure *integer->complex*. Instead, we define how an integer can be transformed into a rational, how a rational is transformed into a real, and how a real is transformed into a complex number. We then allow the system to transform the integer into a complex number through these steps and then add the two complex numbers.

We can redesign our *operate-2* procedure in the following way. For each type, we need to supply a *raise* operator, which "raises" objects of that type one level in the tower. Then when the system is required to operate on two objects of different types, it can successively raise the lower type until the two objects are at the same level in the tower. Exercise 2-55 concerns the details of implementing such a strategy.

Another advantage of a tower is that we can easily implement the notion that every type "inherits" all operations defined on a supertype. For instance, if we do not supply a special procedure for finding the real-part of an integer, we should nevertheless expect that *real-part* will be defined for integers by virtue of the fact that integers are a subtype of complex numbers. In a hierarchy, we can arrange for this to happen by a simple modification to the *operate* procedure given on page 119: If the required operator is not directly defined for the type of the object given, we *raise* the object to its supertype and try again. We thus crawl up the hierarchy, transforming our operand as we go, until we either find a level at which the desired operation can be performed, or we have hit the top, in which case we give up.

A final advantage of a tower (as opposed to a more general hierarchy) is that it gives us a simple way to "lower" a data object to the simplest representation. For example, if we add 2+3i to 4-3i, it would be nice to obtain the answer as the integer 6, rather than as the complex number 6+0i. Exercise 2-56 discusses a way to implement such a lowering operation. (The trick is that we need a general way to distinguish those objects that can be lowered, such as 6+0i, from those which cannot, such as 6+2i.)

Problems with hierarchies

If the data types in our system can be naturally arranged in a tower, then this greatly simplifies the problems of dealing with generic operators on different types, as we have seen. Unfortunately, this is usually not the case. Figure 2-16 illustrates a more complex arrangement of mixed types, this one showing relations between different types of geometric figures. We see that, in general, a type may have more than one subtype. Triangles and quadrilaterals, for instance, are both subtypes of polygons. In addition, a type may have more than one supertype. For example, an isosceles right triangle may be regarded either as an isosceles triangle or as a right triangle. This "multiple supertype" problem is particularly thorny, since it means that there is no unique way to "raise" a type in the hierarchy. Finding the "correct" supertype in which to apply an operator to an object may involve considerable searching through the entire type network on the part of a procedure such as *operate*. Since, in general, there are multiple subtypes for a type, there is a similar problem in coercing a value "down" the type hierarchy. The problem of dealing with large numbers of interrelated types while still preserving modularity in the design of large systems is a very difficult one, and is an area of much current research.

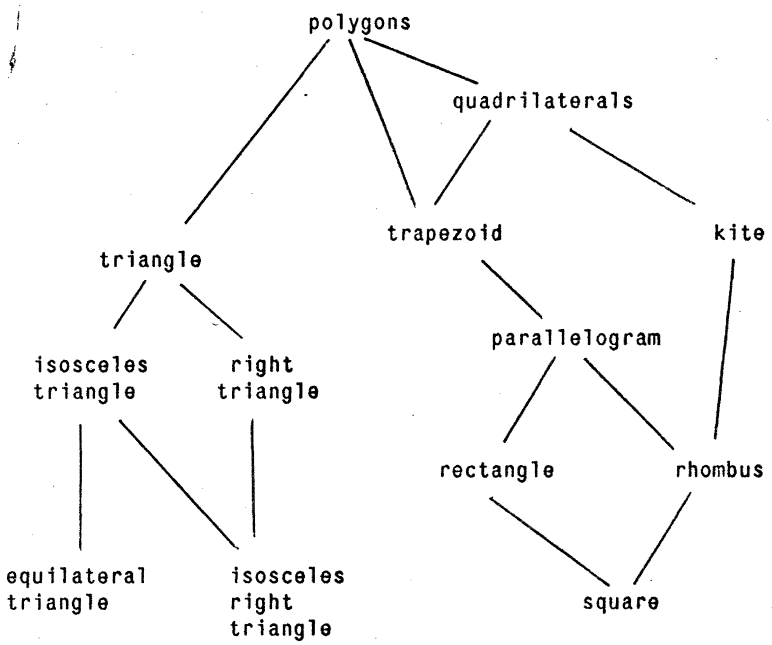


Figure 2-16: Relations between types of geometric figures

Exercise 2-50: The rational arithmetic package of section 2.1.1 can be easily incorporated into our generic arithmetic system. Make the necessary additions and modifications to the package to make it compatible with the conventions of the generic arithmetic system. Install it.

Exercise 2-51: Define a generic equality operator, *equ?* that tests the equality of two numbers and install it as an operator in the generic arithmetic package. It should work for ordinary numbers, rational numbers and complex numbers.

Exercise 2-52: Define and install in the package a generic operator *=zero?*, which tests if its argument is zero.

Exercise 2-53: Suppose we want to include in the package complex numbers whose real parts, imaginary parts, magnitudes, and angles can be either ordinary numbers or rational numbers, or other numbers we might wish to add to the system. Describe and implement the changes to the system needed to accommodate this. You will have to define operators such as *sine* and *cosine* which are generic over ordinary numbers and rationals. You may also need to face some "simplification" issues. For instance, it seems meaningless to have a complex number whose real part is itself complex.

Exercise 2-54: Suppose we are designing a generic arithmetic system for dealing with the following tower of types: integer, rational, real, complex. For each type (except complex), design an appropriate procedure that raises objects of that type one level in the tower. Show how to install a generic *raise* operator that will work for each type (except complex).

Exercise 2-55: Using the *raise* operator of exercise 2-54, describe how to modify the *operate-2* procedure so that it coerces its two operands to have the same type by the method of successive raising, as discussed in section 2.4.2. You will need to devise a way to test which of two types is higher in the tower. Devise a way to accomplish this in a manner which is "compatible" with the rest of the system and, in particular, will not lead to problems in adding new levels to the tower.

Exercise 2-56: We outlined in this section a method for "simplifying" a data object in a tower of types by lowering it in the tower as far as possible. Design a procedure *drop* which accomplishes this for the tower described in exercise 2-54. The key problem is to decide, in some general way, whether an object can be lowered to the next level. For example, the complex number (1/2) + 0i can be lowered as far as *rational*, the complex number 1 + 0i can be lowered as far as *integer*, and the complex number 2 + 3i cannot be lowered at all. Here is a plan for determining whether an object can be lowered: Begin by defining a generic operator *project* that "pushes" an object down in the tower. For example, projecting a complex number would involve throwing away the imaginary part. Then a number can be dropped if, when we *project* it and *raise* the result back to the type we started with, we end up with something equal to what we started with. Show how to implement this idea in detail, by writing a *drop* procedure that drops an object as far as possible. You will need to design the various projection operations, and to install *project* as a generic operator in the system. You will also need to make use of a generic equality operator, such as described in exercise 2-51. Finally, use *drop* to rewrite *operate-2* from exercise 2-55 so that it "simplifies" its answers.

2.4.3. Example: Symbolic Algebra

The manipulation of symbolic algebraic expressions is a complex process which illustrates many of the hardest problems which occur in the design of large-scale systems. An algebraic expression, in general, can be viewed as a hierarchical structure, formed as a tree of operators applied to operands. In fact, many algebraic manipulations can be viewed as recursive tree walks on these expressions. We have already seen an example of this, in the symbolic differentiation program of section 2.2.4.

We can construct algebraic expressions by starting with a set of primitive objects, such as "constants" and "variables," and combining these by means of algebraic operators such as addition and multiplication. As in other languages, we form abstractions that enable us to refer to compound objects in simple terms. Typical abstractions in symbolic algebra are ideas such as "linear combination," "polynomial," "rational function," or "trigonometric function." We can regard these as compound "types" which are often useful for directing the processing of expressions. Our symbolic differentiation program in fact performed just such a dispatch according to whether the expression to be differentiated was a "sum" or a "product."

On the other hand, this type hierarchy is ill-defined. Types like "sum" or "polynomial" are not absolute like "number." They are rather a high-level combining form, a mathematical

glue quite analogous to *cons*. For example, we can form a sum of products. Alternatively, we can form a product of sums. We can also form more complex objects, such as

$$\sin(y^2 + 1)x^2 + \cos(2y)x + \cos(y^3 - 2y^2)$$

which may be described as a "polynomial in x with coefficients which are trigonometric functions of polynomials in y whose coefficients are integers."

We will not attempt to develop here a complete algebraic manipulation system. Such systems are exceedingly complex programs, embodying deep algebraic knowledge and elegant algorithms. What we will do is look at a simple but important part of algebraic manipulation -- the algebra of polynomials. We will illustrate the kinds of decisions that the designer of such a system faces, and how to apply the ideas of abstract data and generic operators to help organize this effort.

Arithmetic on polynomials

Our first problem in designing a polynomial arithmetic system is to decide just what a polynomial is. Polynomials are normally defined relative to certain variables. For simplicity, we will restrict ourselves to polynomials having just one indeterminate (so-called *univariate polynomials*).[22] Usually we define a polynomial to be a sum of terms, each of which is either a coefficient, a power of the indeterminate, or a product of a coefficient and a power of the indeterminate. A coefficient is defined to be an algebraic expression which is not dependent upon the indeterminate of the polynomial. So, for example,

$$5x^2 + 3x + 7$$

is a simple polynomial in x, while

$$(y^2 + 1)x^3 + (2y)x + 1$$

is a polynomial in x whose coefficients are polynomials in y.

Already we are skirting some thorny issues. Is the first of these polynomials the same as the polynomial $5y^2 + 3y + 7$, or not? A reasonable answer might be "yes, if we are considering a polynomial purely as a mathematical function, but no, if we are considering a polynomial to be a syntactic form." The second polynomial is algebraically equivalent to a polynomial in y whose coefficients are polynomials in x. Should our system recognize this, or not? Furthermore, there are other ways to represent a polynomial -- for example as a product of factors, or (for a univariate polynomial) as the set of roots, or as a listing of the values of the polynomial at a specified set of points.[23] We can finesse these questions by insisting that in our algebraic manipulation system, we take a "polynomial" to be a particular syntactic form,

---

[22] On the other hand, we will allow polynomials whose coefficients are themselves polynomials in other variables. This will give us essentially the same representational power as a full multivariate system, although it does lead to coercion problems, as discussed below.

[23] For univariate polynomials, giving the value of a polynomial at a given set of points can be a particularly good representation. Notice that this makes polynomial arithmetic extremely simple, since to obtain, for example, the sum of two polynomials represented in this way, we need only add the values of the polynomials at corresponding points. To transform back to a more familiar representation, we can use the *Lagrange Interpolation Formula*, which shows how to recover the coefficients of a polynomial of degree n given the values of the polynomial at $n + 1$ points.

not its underlying mathematical meaning.

Now we must consider how we go about doing arithmetic on polynomials. In this simple system, we will consider only addition and multiplication. Moreover, we will insist that two polynomials to be combined must have the same indeterminate.

We'll approach the design of our system by following the familiar discipline of data abstraction. We'll assume that a polynomial consists of a variable and a collection of terms, and that we have selectors *variable* and *term-list* which extract those parts from a polynomial. All arithmetic is actually done on the term lists; the variable is just an extension of the type of the polynomial used to check for legitimate polynomial operations. We'll also suppose that we have a constructor *make-polynomial* that assembles a polynomial from a given variable and term list.

The following procedures are the entry points to our polynomial manipulation package:

```
(define (+poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-polynomial (variable p1)
                       (+terms (term-list p1)
                               (term-list p2)))
      (error "Polys not in same var -- +POLY" (list p1 p2))))

(define (*poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-polynomial (variable p1)
                       (*terms (term-list p1)
                               (term-list p2)))
      (error "Polys not in same var -- *POLY" (list p1 p2))))
```

We can now use data-directed programming to install these new procedures in our generic arithmetic system:

```
(put 'polynomial 'add +poly)
(put 'polynomial 'mul *poly)
```

Polynomial addition is performed termwise. Terms of the same order (that is, with the same power of the indeterminate) must be combined. This is done by forming a new term of the same order whose coefficient is the sum of the coefficients of the addends. Terms in one addend for which there is no term of the same order in the other addend simply get accumulated into the sum polynomial being constructed.

In order to manipulate term lists, we'll assume that we have a constructor, *adjoin-term*, which adjoins a new term to a term list. We'll also assume that we have a procedure *empty-termlist?* which tells if a given term list is empty, a *first-term* operator, which extracts the highest order term from a term-list, a *rest-terms* operator which returns all but the highest term. Given a term, we'll suppose that we have selectors *order* and *coeff*, which return, respectively, the order and coefficient of the term. Needless to say, these operators allow us to consider both terms and term lists as data abstractions, whose specific representations we can worry about separately.

Here is the operation that constructs the term list for the sum of two polynomials.[24]

```
(define (+terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1)) (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term (make-term (order t1)
                                          (coeff t1))
                               (+terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term (make-term (order t2)
                                          (coeff t2))
                               (+terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term (make-term (order t1)
                                          (add (coeff t1)
                                               (coeff t2)))
                               (+terms (rest-terms L1)
                                       (rest-terms L2)))))))))
```

The most important point to note here is that we used the generic addition operator *add* to add together the coefficients of the terms being combined. This has powerful consequences, as we will see below.

In order to multiply two term lists, we multiply each term of the first list by all the terms of the other list, repeatedly using a procedure **-term-by-all-terms*, which multiplies a given term by all terms in a given term list. The resulting polynomials (one for each term of the first list) are accumulated into a sum. Multiplying two terms forms a term whose order is the sum of the orders of the factors and whose coefficient is the product of the coefficients of the factors:

```
(define (*terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)  ;this is a procedure that returns an
                            ;empty term list
      (+terms (*-term-by-all-terms (first-term L1) L2)
              (*terms (rest-terms L1) L2))))

(define (*-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (adjoin-term (make-term (+ (order t1) (order t2))
                                (mul (coeff t1) (coeff t2)))
                     (*-term-by-all-terms t1 (rest-terms L))))))
```

---

[24] This operation is very much like the ordered *union-set* operation we developed in Exercise 2-34. In fact, if we think of the terms of the polynomial as a set ordered according to the power of the indeterminate, then the program that produces the term list for a sum is almost identical to *union-set*.

This is really all there is to polynomial addition and multiplication. Notice that, since we operate on terms using the *generic* operators *add* and *mul*, our polynomial package is automatically able to handle any type of coefficient that is known about by the generic arithmetic package. If we include a coercion mechanism such as one of the ones discussed in section 2.4.2 then we also are automatically able to handle operations on polynomials of different coefficient types, such as

$$(3x^2+(2+3i)x+7) * (x^4+(2/3)x^2+(5+3i))$$

Because we installed the polynomial addition and multiplication operators *+poly* and **poly* in the generic arithmetic system as the appropriate *add* and *mul* operators for type *polynomial*, our system is also automatically able to handle polynomial operations such as

$$((y+1)x^2+(y^2+1)x+(y-1)) * ((y-2)x+(y^3+7))$$

The reason is that when the system tries to combine coefficients, it will dispatch through *add* and *mul*. Since the coefficients are themselves polynomials (in *y*) these will be combined using *+poly* and **poly*. The result is a kind of "data-directed recursion" in which a call to, say **poly*, will result in recursive calls to **poly* in order to multiply the coefficients. If the coefficients of the coefficients were themselves polynomials (as we might use to represent polynomials in three variables) the data-direction would assure that the system would follow through another level of recursive calls, and so on through as many levels as the structure of the data dictates.[25]

Representing term lists

Finally, we must confront the problem of implementing a good representation for term lists. Observe that a term list is, in fact, a table of coefficients, indexed by the order of the term. Hence any of the methods for representing sets or tables, as discussed in section 2.2.5, can be applied to this problem. On the other hand, notice that our procedures *+terms* and **terms* always access term lists sequentially from highest to lowest order. Thus we will use some kind of ordered list representation.

How should we structure the list that represents a term list? One consideration is the "density" of the polynomials we intend to manipulate. A polynomial is said to be *dense* if it has nonzero coefficients in terms of most orders. If it has lots of zero terms it is said to be *sparse*. For example:

$$A: \quad x^5 + 2x^4 + 3x^2 - 2x - 5$$

is a dense polynomial, while

$$B: \quad x^{100} + 2x^2 + 1$$

is sparse.

---

[25] In order to make this work completely smoothly, we should also add to our generic arithmetic system the ability to coerce a "number" to a polynomial, by regarding it as a polynomial of degree zero whose coefficient is the number. This is necessary if we are going to perform operations such as

$$(x^2+(y+1)x+5) + (x^2+2x+1)$$

which requires adding the coefficient *y+1* to the coefficient 2.

The term lists of dense polynomials are most efficiently represented as lists of the coefficients. For example, *A* above would be nicely represented as

```
(1 2 0 3 -2 -5)
```

The order of a term in this representation is the *length* of the sublist beginning with that term's coefficient, decremented by one.

Unfortunately, this would be a terrible representation for a sparse polynomial such as *B*. There would be a giant list of mostly zeros, punctuated by a few lonely non-zero terms. Since computer memory is not free, it would be dumb to represent sparse polynomials in this way. A more reasonable representation of the term list of a sparse polynomial is as a list of pairs, each representing a non-zero term. Each pair contains the order of the term and the coefficient for that order. In such a scheme polynomial *B* is efficiently represented as

```
((100 1) (2 2) (0 1))
```

As most polynomial manipulations are performed on sparse polynomials, we will choose to use this method.

Having made this decision, actually implementing the selectors and constructors for terms and term lists is straightforward:

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))

(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
(define (the-empty-termlist) '())

(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

Now only a few minor procedures remain to be defined:

```
(define (make-polynomial variable term-list)
  (attach-type 'polynomial (cons variable term-list)))

(define (variable p) (car p))
(define (term-list p) (cdr p))
```

Exercise 2-57: Install *=zero?* for polynomials in the generic arithmetic package. This will allow *adjoin-term* to work for polynomials with coefficients which are themselves polynomials.

Exercise 2-58: Extend the polynomial system to include subtraction of polynomials.

Exercise 2-59: Define procedures that implement the term list representation described above as appropriate for dense polynomials.

Exercise 2-60: Suppose we want to have a polynomial system that is efficient for *both* sparse and dense polynomials. One way to do this is to allow both kinds of term list representations in our system. The situation is analogous to the complex number example with which we began this chapter, for which

we allowed both rectangular and polar representations. In order to allow this, we must distinguish between different types of term lists, and make the operators on term lists be generic. Redesign the polynomial system to implement this generalization. This is a major effort, not a local change.

Exercise 2-61: A (univariate) polynomial can be divided by another one to produce a polynomial quotient and a polynomial remainder, for example:

$$(x^5 - 1) \div (x^2 - 1) = x^3 + x \quad \text{remainder} \quad x - 1$$

Division can be performed using the standard mechanics taught in high school. That is, divide the high-order term of the dividend by the high-order term of the divisor: the result is the first term of the quotient. Next multiply the result by the divisor and subtract that from the dividend and produce the rest of the answer by (recursively dividing) the difference by the divisor. Stop when the order of the divisor exceeds the order of the dividend and declare the dividend to be the remainder. Also, if the dividend ever reaches zero, return zero as both quotient and remainder.

We can design a */poly* procedure on the model of *+poly* and **poly*. The procedure checks to see if the two polynomials have the same variable. If so, */poly* strips off the variable and passes the problem to a procedure */terms* that performs the division operation on term lists. */Poly* finally re-attaches the variable and the type to the result supplied by */terms*. It is convenient to design */terms* to compute both the quotient and the remainder of a division. */Terms* can take two term lists as arguments and return a list of the quotient term list and the remainder term list.

Fill in the missing expressions to complete the following definition of */terms*, and use this to implement */poly*, which takes two polynomials as arguments and returns a list of the quotient and remainder.

```
(define (/terms tl1 tl2)
  (if (empty-termlist? tl1)
      (list (the-empty-termlist) (the-empty-termlist))
      (let ((t1 (first-term tl1))
            (t2 (first-term tl2)))
        (if (> (order t2) (order t1))
            (list (the-empty-termlist) tl1)
            (let ((new-c (div (coeff t1) (coeff t2)))
                  (new-o (sub (order t1) (order t2))))
              (let ((rest-of-result
                     <compute rest of result recursively>
                     ))
                <form complete result>
                ))))))
```

Type hierarchies in symbolic algebra

Our polynomial system illustrates how objects of one type (polynomials) may in fact be complex objects that have objects of many different types as parts. This poses no real difficulty in defining generic operators. We need only install appropriate generic operators for performing the necessary manipulations of the parts of the compound types. In fact, we saw that polynomials form a kind of "recursive data abstraction," in that parts of a polynomial may themselves be polynomials. Our generic operators and data-directed programming style can handle this complication, without much trouble.

On the other hand, polynomial algebra is an example of a system for which the data types cannot be naturally arranged in a tower. For instance it is possible to have polynomials in *x* whose coefficients are polynomials in *y*. It is also possible to have polynomials in *y* whose coefficients are polynomials in *x*. Neither of these types is "above" the other in any natural way, yet it is often necessary to add together elements from each set. There are several ways to do this. One possibility is to convert one polynomial to the type of the other by expanding and rearranging terms so that both polynomials have the same principal variable. One can impose a tower-like structure on this by ordering the variables, thus always converting any

polynomial to a "canonical form" with the highest priority variable dominant and the lower priority variables buried in the coefficients. This strategy works fairly well except that the conversion may expand a polynomial unnecessarily, making it hard to read and perhaps making it less efficient to work with. The tower strategy is certainly not natural for this domain, nor for any domain where the user can invent new types dynamically using old types in various combining forms, such as trigonometric functions, power series, integrals, and so on.

It should not be surprising that controlling coercion is a serious problem in the design of large-scale algebraic manipulation systems. Much of the complexity of such systems is concerned with relationships among diverse types. Indeed, it is fair to say that we do not yet completely understand coercion, and, in fact, we do not yet completely understand the concept of a data type. Nevertheless, what we know provides us with powerful structuring and modularity principles to support the design of large systems.

Exercise 2-62: By imposing an ordering on variables, extend the polynomial package so that addition and multiplication of polynomials works for polynomials in different variables. (This is not easy!)

Extended Exercise: Rational Functions

We can extend our generic arithmetic system to include *rational functions*. These are "fractions" whose numerator and denominator are *polynomials*, such as

$$\frac{x + 1}{x^3 - 1}$$

The system should be able to add, subtract, multiply, and divide rational functions, performing such computations as

$$\frac{x + 1}{x^3 - 1} + \frac{x}{x^2 - 1} = \frac{x^3 + 2x^2 + 3x + 1}{x^4 + x^3 - x - 1}$$

Observe that the sum has been simplified by removing common factors. (Straightforward "cross multiplication" would have produced a 4th degree polynomial over a 5th degree polynomial.)

If we modify our rational arithmetic package so that it uses generic operators, as suggested in exercise 2-50 then would almost do precisely what we want, *except* for the problem of reducing fractions to lowest terms.

Exercise 2-63: Install generic rational arithmetic (if you haven't already done so), but change *make-rat* so that it does not attempt to reduce fractions to lowest terms. Thus *make-rat* is simply

```
(define (make-rat n d)
  (attach-type 'rational (cons n d)))
```

Test your system by calling *make-rat* on two polynomials, to produce a rational function

```
(define p1 (make-polynomial 'x '((2 1)(0 1))))
(define p2 (make-polynomial 'x '((3 1)(0 1))))
(define rf (make-rat p2 p1))

==>rf
(rational (polynomial x ((3 1)(0 1)))
          (polynomial x ((2 1)(0 1))))
```

and then perform an operation, say, adding this rational function to itself:

```
==>(add rf rf)
(rational (polynomial x ((5 2)(3 2)(2 2)(0 2)))
          (polynomial x ((4 1)(2 2)(0 1))))
```

As the above exercise illustrates, this kind of addition does not reduce fractions to lowest terms. To accomplish this, we do the same thing as with integers, namely, we modify *make-rat* to divide both numerator and denominator by their greatest common divisor.

The notion of "greatest common divisor" makes sense for polynomials. In fact, we can compute the GCD of two polynomials using essentially the same Euclid's algorithm that works for integers:[26]

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

Using this, we could make the obvious modification to define a GCD operation that works on term-lists:

```
(define (gcd-terms a b)
  (if (null? b)
      a
      (gcd-terms b (remainder-terms a b))))
```

where *remainder-terms* picks out the remainder component of the pair returned by the termlist division operation */terms* that was implemented in exercise 2-61

Exercise 2-64: Using */terms*, implement the procedure *remainder-terms* and use this to define *gcd-terms* as above. Now write a procedure *gcd-poly* that computes the polynomial GCD of two polynomials. (The procedure should give an error if the two polynomials are not in the same variable.) Now install in the system a generic operator *greatest-common-divisor* that reduces to the Scheme primitive *gcd* for numbers and to *gcd-poly* for polynomials. Try

```
(define p1 (make-polynomial 'x '((4 1) (3 -1) (2 -2) (1 2))))
(define p2 (make-polynomial 'x '((3 1) (1 -1))))
(greatest-common-divisor p1 p2)
```

What result do you get? Is this correct?

Exercise 2-65: Suppose that *p3* is the polynomial

$$x^5 - 2x^4 + x^3 - x^2 + 2x - 1$$

and that *p4* is

$$x^5 - 2x^4 + 4x^3 - 2x^2 - 5x + 4$$

---

[26] The fact that Euclid's algorithm works for polynomials is formalized in algebra by saying that polynomials form a kind of algebraic domain called a *Euclidean ring*. A Euclidean ring is a domain that admits addition, subtraction, and (commutative) multiplication, together with a way of assigning to each element x of the ring a positive integer "measure" $m(x)$ with the properties (1) $m(xy) \geq m(x)m(y)$ for any non-zero x and y, and (2) given any x and y there exists a q such that $y=qx+r$ and either $r=0$ or $m(r)<m(x)$. From an abstract point of view, this is what is needed to prove that Euclid's algorithm works. For the domain of integers the measure m of an integer is the (absolute value of) the integer itself. For the domain of polynomials, the measure of a polynomial is its degree.

Using *div* demonstrate that these two polynomials are both divisible by $(x-1)^2$ and give the quotient in each case. In fact, $(x-1)^2$ is the greatest-common-divisor of *p3* and *p4*. Now, evaluate the expression

```
(greatest-common-divisor p3 p4)
```

What do you get?

Exercise 2-66: Exercise 2-64 shows that there is a problem with our polynomial GCD program. Explain what is going on. Suggestion: Try tracing *gcd-terms* while computing the example in exercise 2-64. Try performing the division by hand. What problem is the poor program encountering?

We can solve the problem exhibited in exercise 2-64 if we use the following modification of the GCD algorithm (which really works only in the case of polynomials with integer coefficients). Before performing any polynomial division in the GCD computation, we can first multiply the dividend by an integer constant factor, chosen to guarantee that no fractions will arise during the division process. Our answer will differ from the actual quotient by an integer constant factor, but this does not matter in the case of reducing rational functions to lowest terms, because the same integer constant factor will appear in both the numerator and denominator and can be eliminated at the end of the computation.

More precisely, if *p* and *q* are polynomials, let *o1* be the order of *p* (i.e., the order of the largest term of *p*) and let *o2* be the order of *q*. Let *c* be the leading coefficient of *q*. Then it can be shown that, if we multiply *p* by the "integerizing factor" $c^{1+o1-o2}$, the resulting polynomial can be divided by *q* using the */terms* algorithm without introducing any fractions. The operation of multiplying the dividend by this constant and then dividing, is sometimes called the *pseudodivision* of *p* by *q*. The remainder of the division is called the *pseudoremainder*.

So here is how to reduce a rational function to lowest terms:

- Compute the GCD of the numerator and denominator following Euclid's algorithm, but use *pseudo-remainder* rather than *remainder*.
- When you obtain the GCD, multiply both numerator and denominator by the same integerizing factor before dividing through by the GCD, so that division by the GCD will not introduce any non-integer coefficients. As the factor you can use the leading coefficient of the GCD raised to the power *1+o1-o2*, where *o2* is the order of the GCD, and *o1* is the maximum of the orders of the numerator and denominator. This will insure that dividing numerator and denominator by the GCD will not introduce any fractions.
- The result of this operation will be a rational function with integer coefficients. The coefficients will normally be very large, resulting from all of the integerizing factors. So the last step is to remove the redundant factors, by computing the (integer) greatest common divisor of all the coefficients of the numerator and denominator, and dividing through by this factor.

Exercise 2-67: Implement this algorithm as a procedure *reduce*, which takes two term lists *n* and *d* as arguments, and returns a pair *nn*, *dd*, which are *n* and *d* reduced to lowest terms following the algorithm given above. You should, of course, isolate different parts of the computation in different procedures. Some meaningful parts are: a *pseudo-remainder* procedure for term lists, a procedure to compute integerizing factors, a *gcd* procedure for term lists, and a procedure to compute the common (integer) *gcd* of all the coefficients of a polynomial.

Exercise 2-68: Now write a *make-rat-poly* procedure that is analogous to the original *make-rat* for integers, except that it uses your *reduce* to reduce the numerator and denominator to lowest terms. You can now easily obtain a system that handles rational expressions in either integers or polynomials by renaming *make-rat* to *make-rat-number* and defining a new *make-rat* as a *generic* operation that calls *operate-2* with the appropriate numerator/denominator type to dispatch to either *make-rat-poly* or *make-rat-number*. To test your program, try the example at the beginning of this section:

```
==>(define p1 (make-polynomial 'x '((1 1)(0 1))))
==>(define p2 (make-polynomial 'x '((3 1)(0 -1))))
==>(define p3 (make-polynomial 'x '((1 1))))
==>(define p4 (make-polynomial 'x '((1 2)(0 -1))))

==>(define rf1 (make-rat p1 p2))
==>(define rf2 (make-rat p3 p4))

==>(add rf1 rf2)
```

and see if you get the correct answer, correctly reduced to lowest terms.

The GCD computation is at the heart of any system that does operations on rational functions. The algorithm used above, although mathematically simple, is unfortunately extremely slow. The slowness is partly due to the large number of division operations, and partly to the enormous size of the intermediate coefficients generated by the pseudo-divisions. One of the active areas in the development of algebraic manipulation systems is the design of better algorithms for computing polynomial GCDs.[27]

[27] One extremely efficient and elegant method for computing polynomial GCDs was discovered by Richard Zippel in his Ph.D. thesis at MIT [52]. Zippel's method is a probabilistic algorithm, as is the fast test for primality that we discussed in Chapter 1.

Chapter 3
Modularity, Objects, and State

Plus ca change, plus c'est la meme chose.
-- Alphonse Karr

The preceding chapters introduced the basic elements from which programs are made. We saw how primitive procedures 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 structure of a program. In particular, we need to address the problem of designing 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 when constructing 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 adding a new feature, or debugging an old one, will require working 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-oriented* approach and the *stream processing* approach force us to deal with a collection of programming linguistic issues. With objects, we must become concerned with how a computational object can *change* and yet maintain its identity. This raises thorny issues, and in fact will force us to abandon our old substitution model of computation (Chapter 1, section 1.1.5) in favor of a more mechanistic but less theoretically tractable *environment model* of computation. The stream approach, as we shall see, can be most fully exploited when we decouple simulated time in our model from the order of events that take place in the computer during evaluation, which we will accomplish 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. We can characterize an object's state by one or more *state variables*, which among them maintain enough information about past history to determine the object's current behavior. A bank account, for example, has state in that the answer to the question "Can I withdraw one hundred dollars?" depends upon the history of deposit and withdrawal transactions. 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. They influence each other's states 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 a useful view primarily when the state variables of the system can be grouped into closely coupled subsystems which are only loosely coupled to other subsystems.

The *object-oriented* view of a system can be a powerful framework for organizing computational models of the system. For such a simulation to be modular, the simulation program should be decomposed into computational objects that model the actual objects in the system. Each computational object must have its own *local state variables*, which describe the actual object's state. Furthermore, if, in the system being modeled, the states of objects change over time, then, in the computational model, the state variables of the 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, which enables us to change the value associated with a name.[1]

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'll do this using a procedure *withdraw*, which takes an argument *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 dollars in the account, we should obtain the following sequence of responses using *withdraw*.

```
==> (withdraw 25)
75
```

---

[1] Until we introduce the assignment operator, we have no way to construct computational objects whose behavior changes over time. On the other hand, we have implicitly used such objects, *e.g.* in the random number generator used in Chapter 1, in the operator tables of Chapter 2, and in the very fact that we can *define* new values when interacting with the computer.

```
==> (withdraw 25)
50

==> (withdraw 60)
Insufficient funds

==> (withdraw 15)
35
```

Notice that the two expressions *(withdraw 25)*, both executed in the same context, yield *different* values.

To implement *withdraw*, we can use a variable *balance* to indicate the balance of money in the account, and define *withdraw* as a procedure that accesses *balance*. The *withdraw* procedure checks to see if the *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.

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (sequence (set! balance (- balance amount))
                balance)
      "Insufficient funds"))
```

The decrementing done by *withdraw* is accomplished by the expression

```
(set! balance (- balance amount))
```

This uses the *set!* primitive, whose general form is

(set! *⟨name⟩ ⟨new-value⟩*)

Here *⟨name⟩* is a symbol, and *⟨new-value⟩* is any expression. *Set!* changes *name* so that its value is the result of 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]

Withdraw also uses the *sequence* command to specify that the action performed in the case where the *if* test is true should result in evaluating two forms -- first to decrement *balance* and then to return the value of *balance*. In general, evaluating the expression

(sequence $\langle exp_1 \rangle$ $\langle exp_2 \rangle$ . . . $\langle exp_k \rangle$)

causes the expressions $\langle exp_1 \rangle$ through $\langle exp_k \rangle$ to be evaluated in sequence, and the value of

---

[2]The name *set!* reflects a naming convention used in Scheme, that operations that change the values of variables (or that change data structures, as we will see below in section 3.3) are given names that end with an exclamation point. This is similar to the convention of designating predicates by names that end with a question mark.

the final expression $\langle exp_k \rangle$ to be returned as the value of the entire *sequence* form.[3]

While the *withdraw* procedure works as desired, the use of the variable *balance* presents a problem. As specified above, *balance* is a name defined in the global environment, and is freely accessible to be examined or modified by any procedure. It would be much better if we could arrange the program so that *balance* would somehow be *internal* to *withdraw*, so that *withdraw* is the only procedure that can access *balance* directly, and any other procedure can 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:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (sequence (set! balance (- balance amount))
                    balance)
          "Insufficient funds"))))
```

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

Combining *set!* with local variables is the general programming technique that we will use for constructing computational objects with local state. Unfortunately, using this technique raises a serious problem: When we first introduced procedures, we also introduced the substitution model of evaluation in section 1.1.5 to provide an interpretation of what procedure application means. Namely, we said that applying a procedure should be interpreted as evaluating the body of the procedure with the formal parameters replaced by their values. The problem is that, as soon as we introduce assignment into our language, substitution is no longer an adequate model of procedure evaluation. We will see why this is so in section 3.1.2. But as a consequence of this fact, we technically have no way to understand why the *new-withdraw* procedure behaves as claimed above. In order to really understand a procedure such as *new-withdraw*, we will need to develop a new model of procedure evaluation. In section 3.2 we will introduce such a model, together with an

---

[3] We have already used *sequence* implicitly in our programs, because in Scheme, the body of a procedure can be a sequence of forms. Also, the *⟨alternative⟩* part of each clause in a *cond* expression can be a sequence of forms rather than a single form.

[4] In programming language jargon, the variable *balance* is said to be *encapsulated* within the *new-withdraw* procedure. Encapsulation reflects the general system design principle known as the *hiding principle* -- that a system can be made 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."

explanation of *set!* and local variables. First, however, we examine some variations on the theme established by *new-withdraw*.

The following procedure *make-withdraw* creates "withdrawal processors." The formal parameter *balance* in *make-withdraw* specifies the initial amount of money in the account:[5]

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (sequence (set! balance (- balance amount))
                  balance)
        "Insufficient funds")))
```

Make-withdraw can be used as follows to create two objects *W1* and *W2*:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))

==> (W1 50)
50

==> (W2 70)
30

==> (W1 40)
10

==> (W2 40)
Insufficient funds
```

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 represent simple bank accounts. Here is the resulting procedure, which returns a "bank account object" with a specified initial balance.

[5] In contrast to *new-withdraw* above, we do not have to use *let* to make *balance* a local variable, since formal parameters are already local. We will see this more clearly after we have discussed the environment model of evaluation in section 3.2. (See also exercise 3-9.)

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (sequence (set! balance (- balance amount))
                  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT" m))))
  dispatch)
```

Each call to *make-account* sets up an environment with a local state variable *balance*. Within this environment, *make-account* defines two procedures *deposit* and *withdraw* which access *balance*, and an additional procedure *dispatch*, which takes a "message" as input and returns one of the two local procedures. The *dispatch* procedure itself is returned as the value that represents the object. This is precisely the *message passing* style of programming that we saw in section 2-44, although here we are using it in conjunction with the ability to modify local variables.

Make-account can be used as follows:

```
(define acc (make-account 100))

==> ((acc 'withdraw) 50)
50

==> ((acc 'withdraw)) 60)
Insufficient funds

==> ((acc 'deposit) 40)
90

==> ((acc 'withdraw) 60)
30
```

Notice that each call to *acc* returns the locally defined *deposit* or *withdraw* procedure, which is then applied to the specified *amount*. As with *make-withdraw*, another call to *make-account*

```
(define acc2 (make-account 100))
```

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

Exercise 3-1: An *accumulator* is a procedure 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 procedure *make-accumulator* that generates accumulators. The input to *make-accumulator* should specify the initial value of the sum. For example:

```
(define A (make-accumulator 5))

==>(A 10)
15
```

```
==>(A 10)
26
```

Exercise 3-2: In software testing applications, it is useful to be able to count the number of times that a given procedure is called during the course of a computation. Write a procedure *make-monitored*, which takes as input a procedure *f*, which itself takes one input. The result returned by *make-monitored* is a third procedure, say *mf*, which keeps track of the number of times it has been called by maintaining an internal counter. If the input to *mf* is the special symbol *how-many-calls?*, then *mf* returns returns the value of the counter. If the input is the special symbol *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* procedure:

```
(define s (make-monitored sqrt))

==>(s 100)
10

==>(s 'how-many-calls?)
1
```

Exercise 3-3: Modify the *make-account* procedure so that it creates password-protected accounts. That is, *make-account* should take a symbol as an additional argument, as in

```
(define 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 otherwise print a complaint:

```
==> ((acc 'secret-password 'withdraw) 40)
60

==> ((acc 'some-other-password 'deposit) 50)
incorrect password
```

Exercise 3-4: Modify the *make-account* procedure 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 procedure *call-the-cops*.

3.1.2. The Costs of Introducing Assignment

We have seen how the *set!* operation enables us to model objects that have local state. But this advantage comes at a price: Our programming language can no longer be interpreted in terms of the substitution model of procedure evaluation that we introduced in section 1.1.5. Moreover, *no* simple model with "nice" mathematical properties can be an adequate framework for dealing with objects and assignment in programming languages.

To understand why this is true, let's consider a simplified version of the *make-withdraw* procedure of section 3.1.1 that does not bother to check for an insufficient amount.

```
(define (make-simplified-withdraw balance)
    (lambda (amount)
      (set! balance (- balance amount))
      balance))

(define W (make-simplified-withdraw 25))

==>(W 20)
5
```

```
==>(W 10)
-5
```

We'll contrast this procedure with the following *make-decrementer* procedure, which does not use *set!*:

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

Make-decrementer returns a procedure that subtracts its input from a designated amount *balance*, but there is no accumulated effect over successive calls, as with *make-simple-withdraw*:

```
(define D (make-decrementer 25))

==> (D 20)
5

==> (D 10)
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*, which reduces the expression to

```
((lambda (amount) (- 25 amount)) 20)
```

Now we apply the operator by substituting 20 for *amount* in the body of the *lambda* expression:

```
(- 25 20)
```

and 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 body of *make-simplified-withdraw*, which reduces the expression to

```
((lambda (amount) (set! 25 (- 25 amount)) 25) 20)
```

Now we apply the operator by substituting 20 for *amount* in the body of the *lambda* expression:

```
((set! 25 (- 25 20)) 25)
```

If we adhered to the substitution model, we would have to say that the meaning of the procedure application is to first set 25 to 5 and then return 25 as the value of the expression. This makes no sense at all.

The problem 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 *set!*, and the

idea that the value of a variable can *change*, then a variable can no longer be simply a name. Rather, a variable must now somehow refer to a *place* where a value can be stored, and the value stored at this place can change. In section 3.2 below, we'll see how environments play this role of "place" in our computational model.

Sameness and change

The issue surfacing here is more profound than merely the breakdown of a particular model of computation. As soon as we introduce change into our computational models, many notions which were previously straightforward now become problematical. As an important example, let's consider the concept of two things being "the same."

Suppose we create two procedures by calling the above *make-decrementer* procedure twice with the same argument:

```
(define D1 (make-decrementer 25))

(define D2 (make-decrementer 25))
```

Now we ask: Are *D1* and *D2* the same? An acceptable answer is to say yes, because *D1* and *D2* have the same computational behavior -- each is a procedure that subtracts its input from 25. In fact, *D1* could be substituted for *D2* in any computation without changing the result. Another way to justify considering *D1* and *D2* to be the same is to observe that, in the substitution view of computation, *D1* and *D2* are both names for the same expression *(make-decrementer 25)*.

Contrast this with making two calls to *make-simplified-withdraw*:

```
(define W1 (make-simplified-withdraw 25))

(define 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 sequence of interactions:

```
==> (W1 20)
5

==> (W1 20)
-15

==> (W2 20)
5
```

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" without changing the values of expressions is said to be *referentially transparent*. Referential transparency is violated when we include *set!* 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, and programs with assignment are susceptible to bugs that cannot occur in the types

of programs we have been dealing with up until now.

Once we forgo referential transparency, the notion of when two computational objects are "the same" becomes difficult to capture in a formal way. Indeed, the meaning of "the same" in the real world which our programs model is hardly clear in itself. For in general, we can only determine that two apparently identical objects are indeed "the same one" by modifying one object and then observing the other object to see if it has changed in the same way. But what do we mean by "change"? Our usual notion of change is to say that some property of an object "has changed" if we have observed that same object twice with the particular property differing in the two observations. So we cannot say that an object "has changed" unless we can determine that we are looking at "the same" object before and after the change, and we cannot determine sameness without observing the effects of change.

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

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

versus modeling the situation as

```
(define peter-acc (make-account 100))
(define 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. But in the second situation, 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, namely the bank account, that has *two* different names -- *peter-acc* and *paul-acc*. For, 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*. With reference to the previous paragraph's remarks on "sameness" and "change," observe that, if Peter and Paul could only examine their bank balances, and not perform operations that changed the balance, then the issue of whether the two accounts are distinct would be moot.[6]

Exercise 3-5: Consider the bank account objects created by *make-account*, with the associated password protection modification described in exercise 3-3. Suppose that our banking system requires the ability to make joint accounts. Define a procedure *make-joint* that accomplishes this. *Make-joint* should take three arguments. The first is a password-protected account. The second must match the password with which the account was defined in order for the *make-joint* operation to proceed. The third input is a new password. *Make-joint* is to create an additional access to the

[6] The phenomenon of a single computational object being accessed by more than one name is known as *aliasing*. The joint bank account situation is 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 [27, 35].

original account using the new password. For example, if *peter-acc* is a bank account with password *open-sesame*, then

```
(define paul-acc (make-joint peter-acc
                             'open-sesame
                             'rosebud))
```

will allow one to make transactions on *peter-acc* using the name *paul-acc* and the password *rosebud*. You may wish to modify your solution to exercise 3-3 to accommodate this new feature.

Exercise 3-6: When we defined the substitution model of evaluation in section 1.1.5, we said that the first step in evaluating an expression is to evaluate the subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left-to-right vs. right-to-left). When we introduce assignment, the order in which the arguments to a procedure are evaluated can make a difference to the result. Define a simple procedure *f* with the property that evaluating

```
(+ (f 0) (f 1))
```

will return 0 if the arguments to + are evaluated in left-to-right order, but will return 1 if the arguments are evaluated in right-to-left order.

3.1.3. The Benefits of Introducing Assignment

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, let us consider the design of a random number generator. This is to be a procedure *rand* that, whenever it is called, returns an integer chosen at random.

Of course, 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, assume that we have a procedure, called *rand-update*, which 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 x_1, x_2, x_3, ..., will have the desired statistical properties.[7]

We can implement *rand* as a procedure with a local state variable *x* which 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. Knuth [24] Chapter 3 includes an extensive discussion of techniques for generating sequences of random numbers and establishing their statistical properties. Notice that the *rand-update* procedure 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 problems in mathematics as well as philosophy. Kolmogorov, Solomonoff, and Chaitin have made great progress in clarifying these problems. A discussion of these issues can be found in [5].

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

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, let's 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 based upon 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; i.e., that their greatest common divisor will be equal to 1.[8] To obtain the approximation to π, we perform a large number of experiments. In each experiment we choose two random integers and perform a test to see if their *gcd* is equal to 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:

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))

(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (-1+ trials-remaining) (1+ trials-passed)))
          (else
           (iter (-1+ trials-remaining) trials-passed))))
  (iter trials 0))
```

Now let's 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:

```
(define (estimate-pi trials)
  (sqrt (/ 6
           (random-gcd-test trials random-init))))
```

[8] This theorem is due to E. Cesaro (1881). See Knuth [24] section 4.5.2 for a discussion and a proof.

```
(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond
         ((= trials-remaining 0)
          (/ trials-passed trials))
         ((= (gcd x1 x2) 1)
          (iter (-1+ trials-remaining) (1+ trials-passed) x2))
         (else
          (iter (-1+ trials-remaining) trials-passed x2))))))
  (iter trials 0 initial-x))
```

Although this is still a simple program, it betrays some painful breaches of modularity. In our first version of the program, using *rand*, we are able to express the Monte Carlo method directly as a general *monte-carlo* procedure which takes as a parameter an arbitrary *experiment* procedure. 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 procedure *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 abstract out the Monte-Carlo idea as one which can be applied to other problems. In the first version of the program, assignment encapsulates the state of the random number generator within the *rand* procedure, so that the details of random number generation remain independent of the rest of the program.

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 we could if all state had to be manipulated explicitly as parameters. Unfortunately, the story is not so simple. In section 3.4 we will see how the programming technique of *stream processing* enables us to recover much of our lost modularity without introducing assignment and its concomitant problems as indicated in section 3.1.2. But to do so, we must adopt a very different perspective on objects and, most strikingly, a different perspective on *time* in our computer programs. We will return to this discussion in section 3.4. First, however, we address the issue of providing a computational model for program expressions that involve assignment, and explore the uses of objects in designing simulations.

Exercise 3-7: 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* procedure which is called with an argument that is either the symbol *generate* or the symbol *reset* as follows.

```
(rand 'generate)
```

produces a new random number.

```
((rand 'reset) <new-value>)
```

resets the internal state variable to the designated *new-value*. Thus by resetting the state, one can generate repeatable sequences. These are very handy to have when testing and debugging programs

that use random numbers.

3.2. The Environment Model of Evaluation

When we introduced compound procedures in Chapter 1, we used the substitution model of evaluation (Section 1.1.5) to define what is meant by applying a procedure to arguments:

- To apply a compound procedure to arguments, evaluate the body of the procedure 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 3.1.2 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 pair variables 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, that 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.

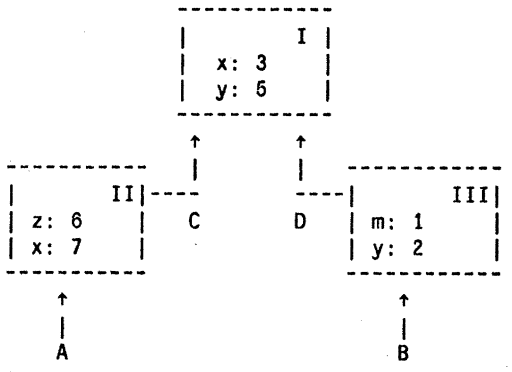


Figure 3-1: A simple environment structure.

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 plays a crucial part in 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 *(+ 1 1)* depends on an understanding that one is operating in a context in which + is the symbol for addition and (the numeral) "1" is the symbol for (the number) 1. Thus, in our model of evaluation, we will always speak of evaluating an expression with respect to some environment. To describe interactions with the Lisp interpreter, we will suppose that there is a *global environment*, consisting of a single frame (with no enclosing environment) that includes values for the symbols associated with the primitive procedures. For example, the idea that + is the symbol for addition is captured by saying that the symbol + is bound in the global environment to the primitive addition procedure.

3.2.1. The Rules for Evaluation

The overall specification of how the interpreter evaluates a combination remains the same as we first introduced it in section 1.1.3:

To evaluate a combination (other than a special form):

1. Evaluate the subexpressions of the combination.
2. Apply the value of the operator subexpression to the values of the operand subexpressions.

The new element provided by the environment model of evaluation is to specify what it means to apply a compound procedure to arguments.[9]

In the environment model of evaluation, a procedure is always a *pair* consisting of some *code* and a pointer to an *environment*. Procedures are created in one way only: by evaluating a `lambda` expression. This produces a procedure whose code is obtained from the text of the `lambda` expression and whose environment is the environment in which the `lambda` expression was evaluated to produce the procedure.

For example, consider the procedure definition

```
(define (square x)
  (* x x))
```

evaluated in the global environment. The procedure definition syntax is just "syntactic sugar" for an underlying implicit `lambda` expression. In terms of the evaluation model, it would be equivalent to have typed in the expression

---

[9] Assignment also introduces a subtlety into step 1 of the evaluation rule. As shown in exercise 3-6, the presence of assignment allows us to write expressions that will produce different values, depending on the order in which the subexpressions in a combination are evaluated. Thus, to be precise, we should specify an evaluation order in step 1 (e.g., left-to-right or right-to-left). However, this order should always be considered to be an implementation detail, and one should never write programs that depend on some particular order. For instance, a compiler should be able to feel free to optimize a program by varying the order in which subexpressions are evaluated.

```
(define square
  (lambda (x) (* x x)))
```

which instructs the computer to evaluate the expression

```
(lambda (x) (* x x))
```

and bind the variable *square* to the resulting value, all in the global environment.

Figure 3-2 shows the result of evaluating this expression. The procedure object is a pair whose code specifies that the procedure has one formal parameter, namely *x*, and a procedure body *(* x x)*. The environment part of the procedure is a pointer to the global environment, since that is the environment in which the *lambda* expression was evaluated to produce the procedure. The entire procedure object is now bound to the variable *square* in the global environment. This means that *square* is added to the global frame, with a binding to the procedure object. In general, *define* creates definitions by adding bindings to frames.

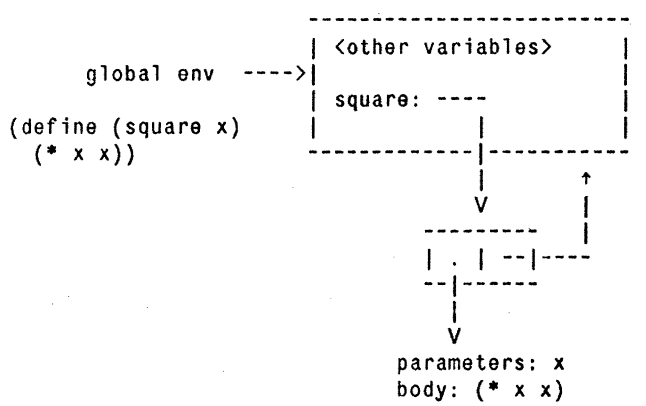


Figure 3-2: Result of evaluating *(define (square x) (* x x))* in the global environment.

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

Figure 3-3 shows the environment structure created by evaluating the expression *(square 5)* in the global environment, where *square* is the procedure generated in figure 3-2. Applying the procedure 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 procedure, is bound to the argument 5. The pointer leading up from this frame shows that its enclosing environment is the global environment, since that is the environment that is part of the *square* procedure. Within *E1*, we evaluate the body of the procedure *(* x x)*. Since the value of *x* in *E1* is 5, the result is *(* 5 5)*, or 25.

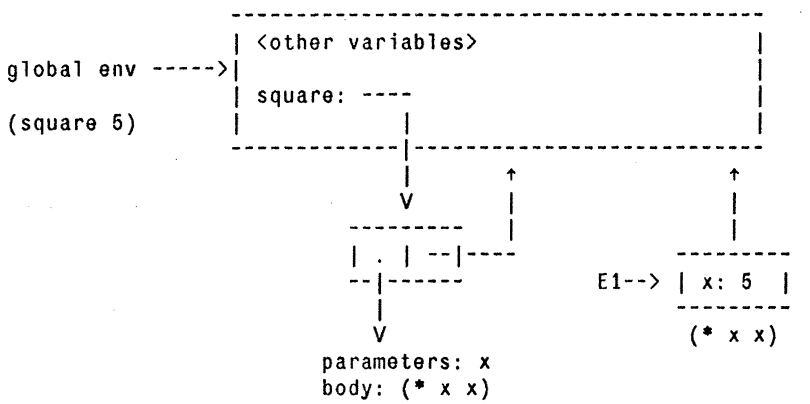


Figure 3-3: Environment created by evaluating *(square 5)* in the global environment.

We can summarize the environment model of procedure application by two rules:

- *Rule 1:* A procedure object is applied to a set of arguments by constructing a frame, binding the formal parameters of the procedure to the actual arguments of the call, and then evaluating the body of the procedure in the context of the new environment constructed. The new frame has as its enclosing environment the environment part of the procedure object being applied.
- *Rule 2:* A procedure is created by evaluating a *lambda* expression relative to a given environment as follows: a new procedure object is formed, combining the text of the *lambda* expression with a pointer to the environment in which the procedure was created.

We also specify that defining a symbol using *define* creates a binding in the current environment frame, which assigns to the symbol the indicated value.[10]

Finally, we specify the behavior of *set!*, the operation that forced us to introduce the environment model in the first place. Evaluating the form

(set! *<variable>* *<value>*)

in some environment locates the binding of the variable in the environment and changes that binding to indicate the new value. That is to say, one finds the first frame in the environment that contains a binding for the variable, and modifies that frame. If the variable is unbound in the environment, then *set!* signals an error.

These evaluation rules, while considerably more complex than the substitution model, are still reasonably straightforward. Moreover, the evaluation model, although abstract, provides a correct description of how the interpreter evaluates expressions, and in Chapter 4 we shall see how this model can serve as a blueprint for constructing a working interpreter. The

---

[10] If there is already a binding for the variable in the current frame, then the binding is changed. This is convenient because it allows redefinition of symbols, but it also means that *define* can be used to change values, thus bringing up the problems of assignment without explicitly using *set!*. Because of this, some people prefer redefinitions of existing symbols to signal errors or warnings.

following sections work through the details of the model in analyzing some illustrative programs.

3.2.2. Evaluating Simple Procedures

When we introduced the substitution model in section 1.1.5, we showed how the combination

```
(f 5)
```

evaluates to 136, given the following procedure definitions:

```
(define (square x)
  (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

We can analyze the same example using the environment model. Figure 3-4 shows the three procedure objects created by evaluating the definitions of *f*, *square*, and *sum-of-squares* in the global environment. Each procedure object consists of a pointer to some code, together with a pointer to the global environment.

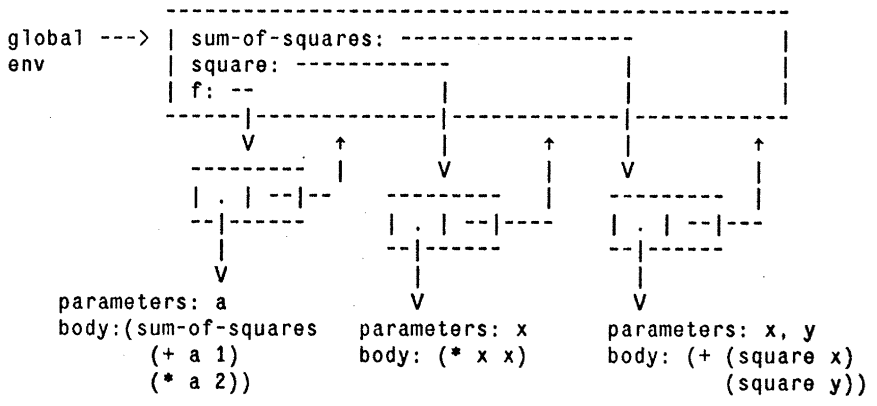


Figure 3-4: Procedure objects in the global frame.

In figure 3-5 we see the environment structure created by evaluating the expression *(f 5)*. The call to *f* creates a new environment *E1* beginning with a frame in which *a*, the formal parameter of *f*, is bound to the argument 5. In *E1*, we evaluate the body of *f*:

```
(sum-of-squares (+ a 1)
                (* a 2))
```

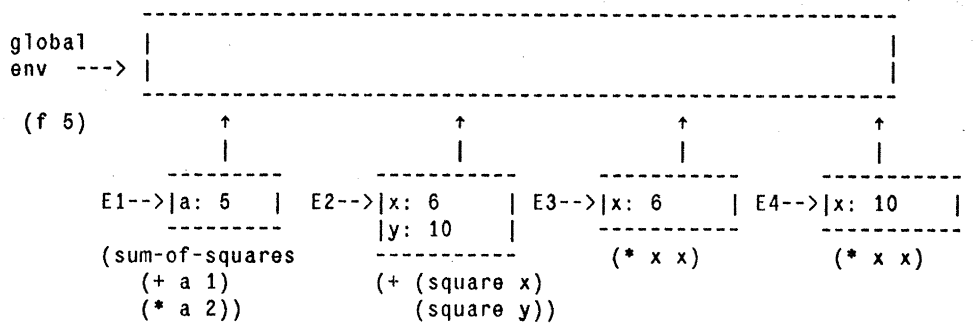


Figure 3-5: Environments created by evaluating *(f 5)* using the procedures in figure 3-4.

To evaluate this combination, we first evaluate the subexpressions. The first subexpression, *sum-of-squares*, has a value that is a procedure 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 containing environment, i.e., to the global environment, and find the binding shown in figure 3-4.) The other two subexpressions are evaluated by applying the primitive operations + and * to evaluate the combinations *(+ a 1)* and *(* a 2)* to obtain 6 and 10, respectively.

Now we apply the procedure object *sum-of-squares* to the arguments 6 and 10. This results in a new environment *E2* in which the formal parameters *x* and *y* are bound to the arguments. Within *E2* we evaluate the combination

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

This leads us to evaluate *(square x)*, where *square* is found in the global frame, and *x* is 6. Once again, we set up an 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 evaluating *sum-of-squares*, we must evaluate the subexpression *(square y)*, where *y* is 10. This *second* call to *square* creates *another* environment *E4*, in which *x*, the formal parameter of *square*, is bound to 10. And within *E4* we must evaluate *(* x x)*.

The important point to observe is that *each* call to *square* creates a new environment containing a binding for *x*. We can see here how the different frames serve to keep separate the different local variables all named *x*. Notice that each frame created by *square* points to the global environment, since this is the environment indicated by the *square* procedure object.

Once all the subexpressions have been evaluated, the results are returned. The two calls to *square* generate values which are added by *sum-of-squares*, and this result is returned to *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 3-8: In section 1.2.1, we used the substitution model to analyze two procedures for computing factorials, a recursive version

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

and an iterative version

```
(define (factorial n)
  (fact-iter 1 1 n))

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

Show the environment structures created during the computation of *(factorial 6)* by each of these procedures.[11]

3.2.3. Frames as the Repository of Local State

We can use the environment model to explain how procedures and assignment can be used to represent objects with local state. As an example, we consider a "withdrawal processor" from section 3.1.1, created by calling the procedure:

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (sequence (set! balance (- balance amount))
                  balance)
        "Insufficient funds")))
```

Let us describe the evaluation of

```
(define W1 (make-withdraw 100))
```

followed by

```
==> (W1 50)
50
```

Figure 3-6 shows the result of defining the *make-withdraw* procedure in the global environment. This produces a procedure object that contains a pointer to the global environment. So far, this is no different from the examples we have already seen, except that, in this case, the *body* of the procedure code is itself a *lambda* expression.

[11] The environment model will *not* clarify our claim in section 1.2.1 that the interpreter can execute a procedure such as *fact-iter* in a constant amount of space using *tail recursion*. We will discuss tail recursion when we deal with the control structure of the interpreter in Chapter 5, section 5.2.i

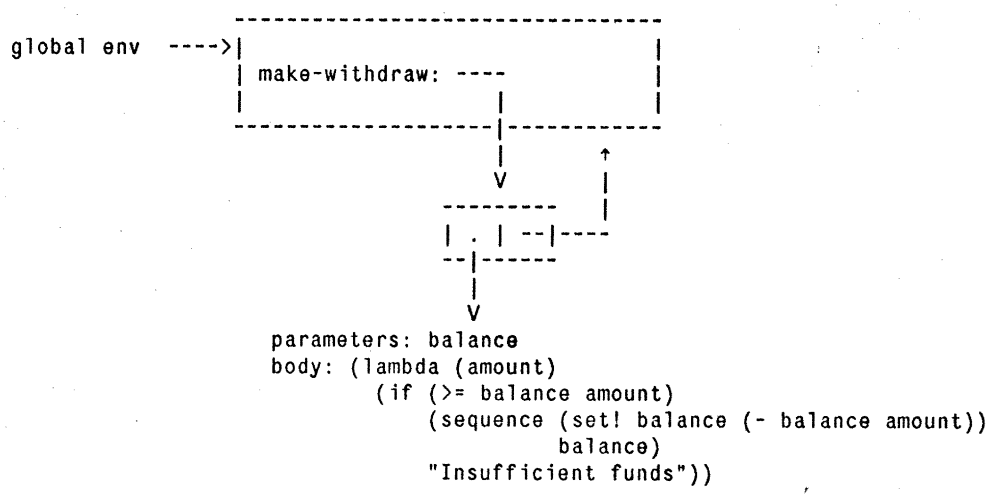


Figure 3-6: Result of defining *make-withdraw* in the global environment.

The interesting part of the computation happens when *make-withdraw* is applied to an argument:

```
(define W1 (make-withdraw 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*, namely, the *lambda* expression. This constructs a new procedure object, whose code is as specified by the *lambda*, and whose environment is *E1*, the environment in which the *lambda* was evaluated to produce the procedure. The resulting procedure object is the value returned by the call to *make-withdraw*. This is bound to *W1* in the global environment, since the *define* itself is being evaluated in the global environment. Figure 3-7 shows the resulting environment structure.

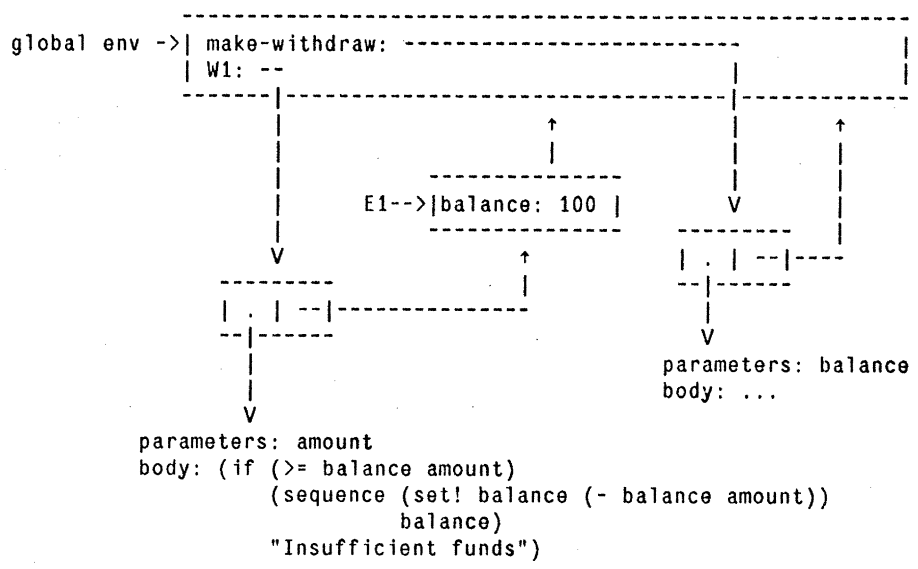


Figure 3-7: Result of *(define W1 (make-withdraw 100))*.

Now we can analyze what happens when *W1* is applied to an argument:

```
==> (W1 50)
50
```

We begin by constructing a frame in which *amount*, the formal parameter for *W1*, is bound to the argument 50. The crucial point to observe is that this frame has as its containing frame, not the global environment, but rather the environment *E1*, because this is the environment that is specified by the *W1* procedure object. Within this new environment, we evaluate the body of the procedure

```
(if (>= balance amount)
    (sequence (set! balance (- balance amount))
              balance)
    "Insufficient funds")
```

The resulting environment structure is shown in figure 3-8. Observe that the expression being evaluated references both *amount* and *balance*. *Amount* will be found in the first frame in the environment, while *balance* will be found by following the containing pointer to *E1*.

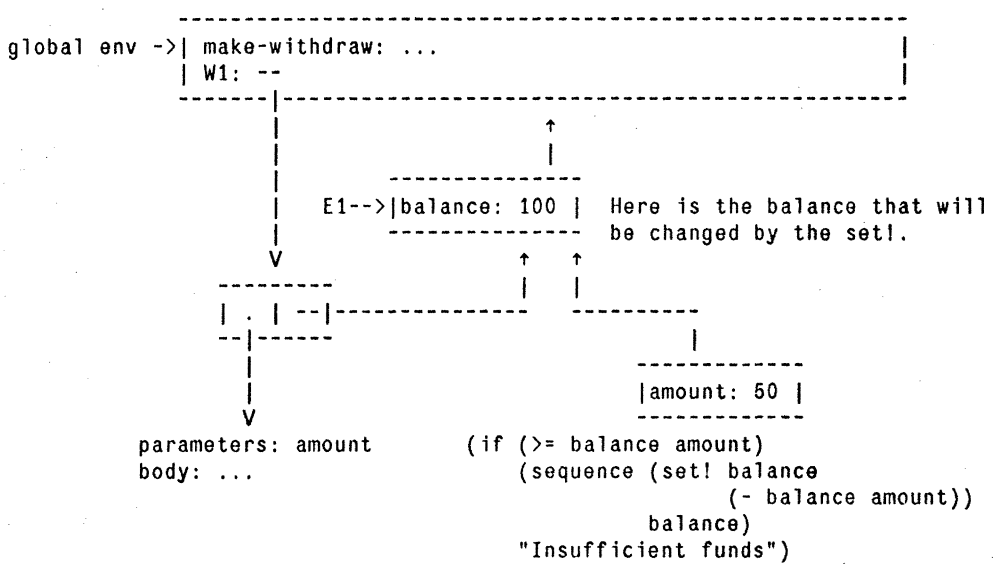


Figure 3-8: Environments created by applying the procedure object *W1*.

When the *set!* instruction is executed, the binding of *balance* in *E1* is changed. At the completion of the call to *W1*, *balance* is now 50, and the frame that contains *balance* is still pointed to by the procedure object *W1*. The local frame in which we executed the code that changed *balance* is no longer relevant, since the procedure 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 containing environment is *E1*. We see that *E1* serves as the "place" that holds the local state variable for the procedure object *W1*. Figure 3-9 shows the situation after the call to *W1*.

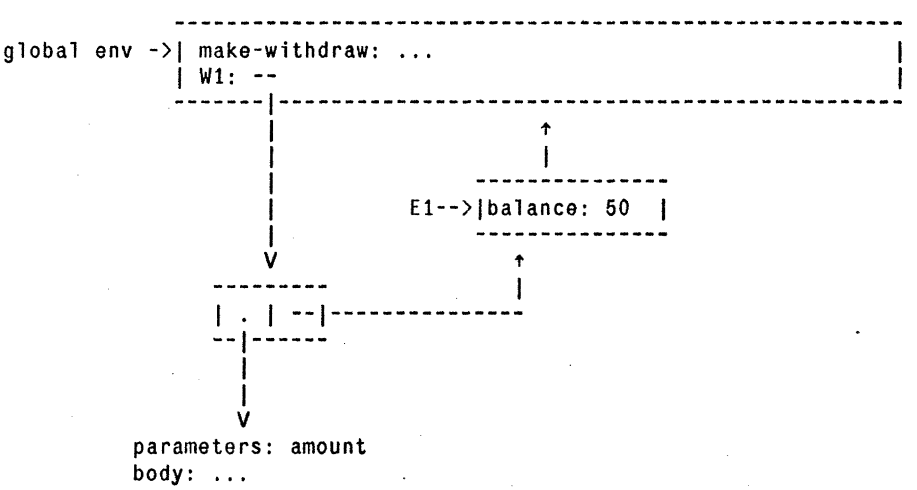


Figure 3-9: Result of calling *W1* in figure 3-8.

Observe what happens when we create a second "withdraw" object by making another call

to *make-withdraw*:

```
(define W2 (make-withdraw 100))
```

This produces the environment structure shown in figure 3-10.

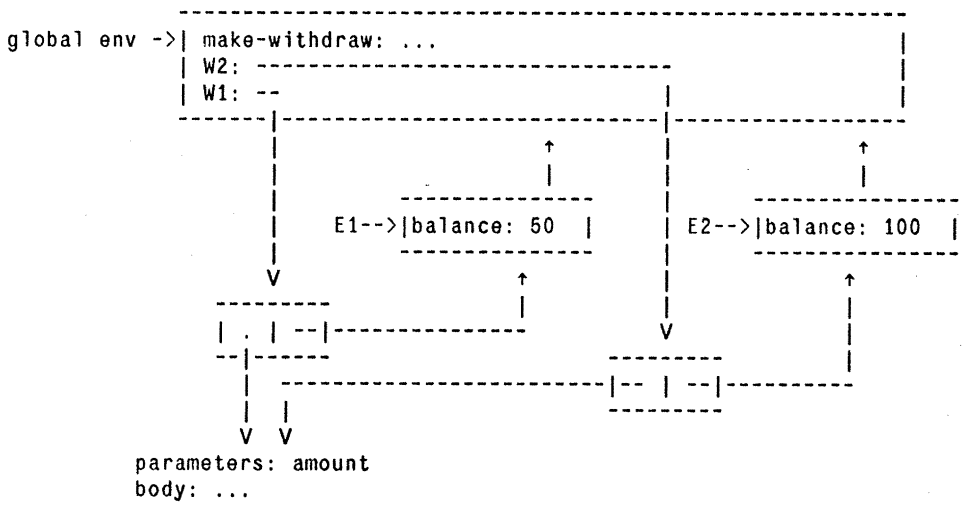


Figure 3-10: Using *(define W2 (make-withdraw 100))* to create a second object.

The figure shows that *W2* is a procedure object; *i.e.*, 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* share the same code; *i.e.*, the code of the *lambda* expression in the body of *make-withdraw*.[12] We see here why *W1* and *W2* behave as independent objects. Calls to *W1* reference the state variable *balance* stored in *E1* while calls to *W2* reference the *balance* stored in *E2*. Thus changes to the local state of one object will not affect the other object.

Exercise 3-9: In the *make-withdraw* procedure, 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:

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (sequence (set! balance (- balance amount))
                    balance)
          "Insufficient funds"))))
```

Recall from section 1.3.2 that *let* is simply "syntactic sugar" for a procedure call:

```
(let ((<var> <exp>)) <body>)
```

is interpreted as an alternate syntax for

```
((lambda (<var>) <body>) <exp>)
```

---

[12] Whether or not *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.

Use this fact to analyze with the environment model the behavior of the alternate version of *make-withdraw*, drawing figures like the ones above to illustrate the interactions

```
(define W1 (make-withdraw 100))

(W1 50)

(define W2 (make-withdraw 100))
```

Show that the two versions of *make-withdraw* create objects with the same behavior. How do the environment structures for the two versions of *make-withdraw* differ?

3.2.4. Internal Definitions

In section 1.1.8 we introduced the idea that procedures can have internal definitions, thus leading to a "block structure" as in the following procedure to compute square roots.

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) .001))

  (define (improve guess)
    (average guess (/ x guess)))

  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))

  (sqrt-iter 1)))
```

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 procedure *good-enough?* has been called for the first time with *guess* equal to 1.

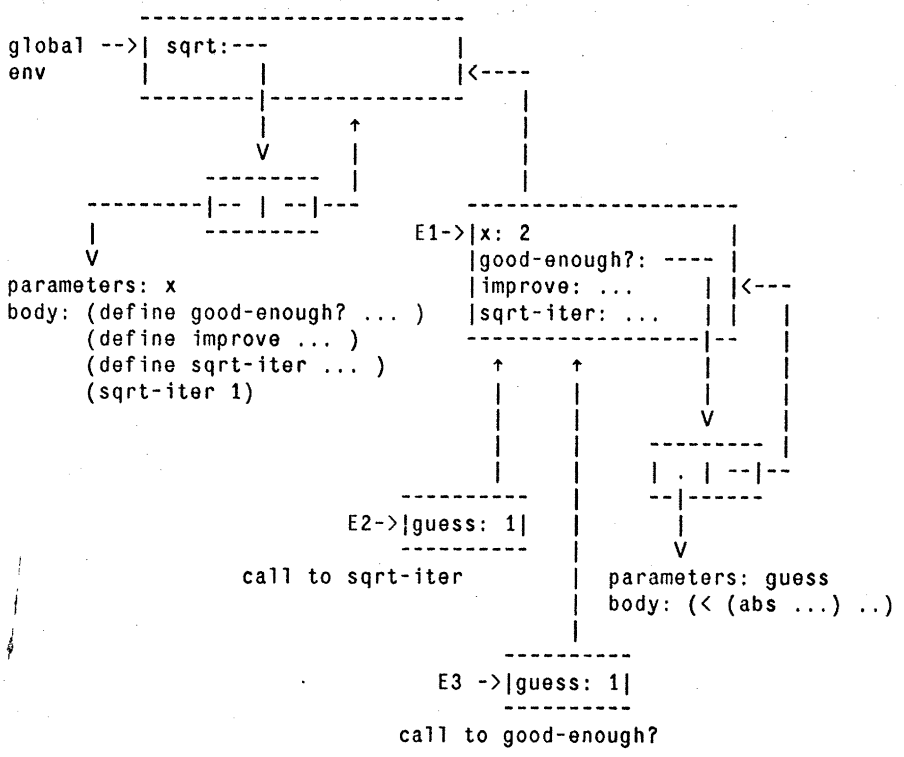


Figure 3-11: *Sqrt* procedure with internal definitions.

Observe the structure of the environment. *Sqrt* itself is a symbol in the global environment that is bound to a procedure object whose associated environment is the global environment. When *sqrt* was called, it formed a new environment *E1*, subordinate to the global environment, in which the parameter *x* is bound to 2. The body of *sqrt* is now evaluated in *E1*. Since, the first expression in the body of *sqrt* is

```
(define (good-enough? guess)
  (< (abs (- (square guess) x)) .001))
```

evaluating this expression defines the procedure *good-enough?* in the environment *E1*. More precisely, the symbol *good-enough?* is added to the first frame of *E1*, and *good-enough?* is bound to a procedure object whose associated environment is *E1*. Similarly, *improve* and *sqrt-iter* are defined as procedures in *E1*. For conciseness, figure 3-11 shows only the procedure object for *good-enough?*.

After the local procedures are defined, *sqrt* executes the expression

```
(sqrt-iter 1)
```

still in environment *E1*. So the procedure object bound to *sqrt-iter* in *E1* is called with 1 as an argument. This results in creating a new environment *E2*, as shown in figure 3-11, in which *guess*, the parameter of *sqrt-iter*, is bound to 1. *Sqrt-iter* in turn calls *good-enough?* with the value of *guess* (from *E2*) as the parameter for *good-enough?*. This sets up another environment *E3* in which *guess* (the parameter of *good-enough?*) is bound to 1. Notice that although *sqrt-iter* and *good-enough?* both have a parameter named *guess*, these are two distinct local variables, located in different frames. Observe also that that *E2* and *E3* both

have *E1* as their enclosing environment, because the *sqrt-iter* and *good-enough?* procedures 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* procedure is called.

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

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

Exercise 3-10: In section 3.2.3, we saw how the environment model explained the evaluation of procedures with local state. Now we have seen how internal definitions work. A typical message passing procedure contains both of these aspects. Consider the message passing styled bank account procedure of section 3.1.1:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (sequence (set! balance (- balance amount))
                  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT")
                m)))
  dispatch)
```

Show the environment structure generated by the sequence of interactions

```
(define acc (make-account 50))

==> ((acc 'deposit) 40)
90

==> ((acc 'withdraw) 60)
30
```

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

```
(define acc2 (make-account 100))
```

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

3.3. Modeling with Mutable Data

Chapter 2 dealt with compound data as a means for constructing computational objects that have several parts, in order to model real-world objects that have several aspects. In that chapter, we introduced the discipline of *data abstraction*, according to which data structures are specified in terms of *constructors*, that create data objects, and *selectors*, that access the parts of compound data objects. But we now know that there is another aspect of data that Chapter 2 did not address. We have learned that 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. Modeling a banking system, for instance, 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*.

We saw in Chapter 2 that Lisp provides pairs as a general-purpose "glue" for synthesizing compound data. We begin this section by defining basic mutators for pairs, so that pairs can now serve as building blocks for constructing mutable data objects. These mutators greatly enhance the representational power of pairs, enabling us to build data structures other than the sequences and trees that we worked with in section 2.2. We also present some examples of simulations in which complex systems are modeled as collections of objects with local state.

3.3.1. Mutable List Structure

The basic operations on pairs -- `cons`, `car`, and `cdr` -- can be used to construct list structure and to select parts from lists, but they are incapable of changing existing list structure. The same is true of the other list operations we have used so far, such as `append` and `list`, since these can be defined in terms of `cons`, `car`, and `cdr`. In order to modify list structures we need new operations.

The primitive mutators for list structure are `set-car!` and `set-cdr!`. `Set-car!` takes two arguments, the first of which must be a pair. It modifies this pair, replacing the `car` pointer by a pointer to the second argument of `set-car!`. As an example, suppose that *x* is bound to the list `((a b) c d)` and *y* to the list `(e f)` as illustrated in figure 3-12.

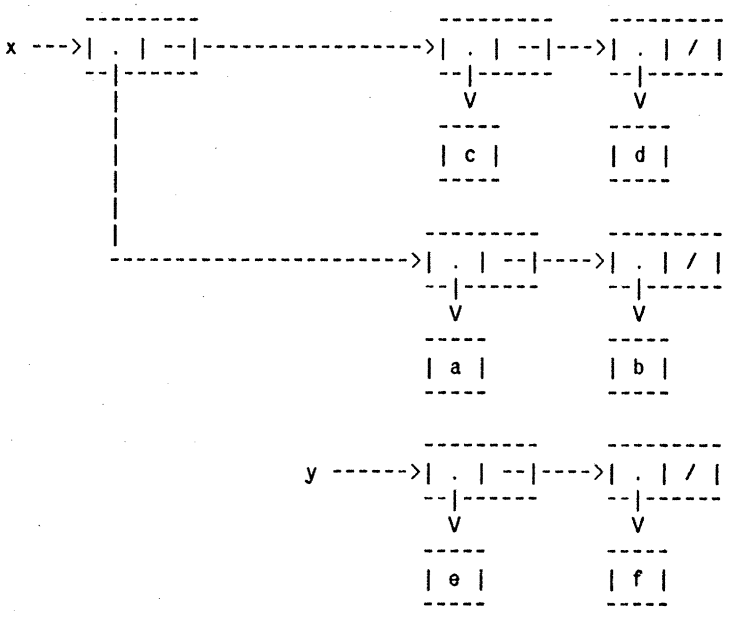


Figure 3-12: Lists *x*: *((a b) c d)* and *y*: *(e f)*

Evaluating the expression *(set-car! x y)* modifies the pair to which *x* is bound, replacing its *car* pointer by a pointer to *y*. The result of the operation is shown in figure 3-13. The structure *x* has been modified, and would now be printed by the interpreter as *((e f) c d)*. The pairs representing the list *(a b)*, identified by the pointer that was replaced, are now detached from the original structure.[13]

[13] We see from this that mutation operations on lists can create "garbage" that is not part of any needed structure. When we discuss Lisp implementations in Chapter 5, we will see in section 5.3 that Lisp memory management systems include a *garbage collector*, which identifies and recycles the memory space used by unneeded pairs.

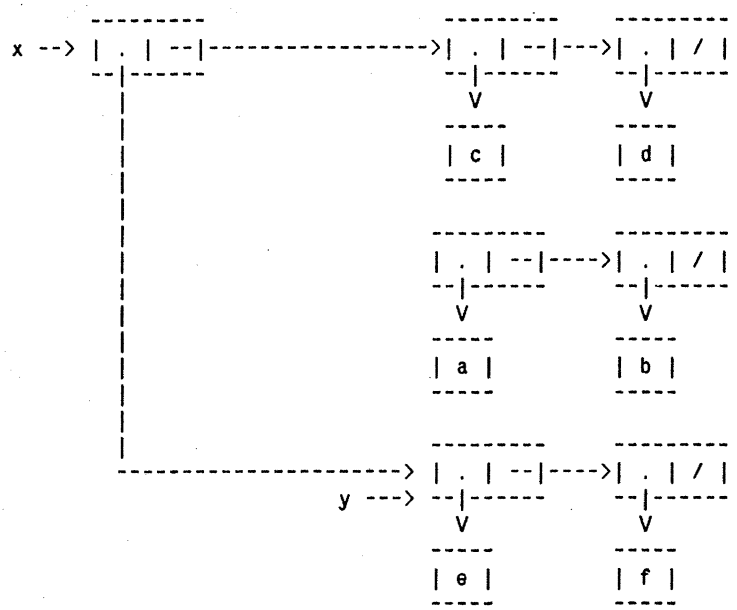


Figure 3-13: Effect of *(set-car! x y)* on the lists in figure 3-12.

It is instructive to compare figure 3-13 with figure 3-14, which illustrates the result of executing

```
(define z (cons y (cdr 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 *cons* operation, and the list bound to *x* is unchanged.

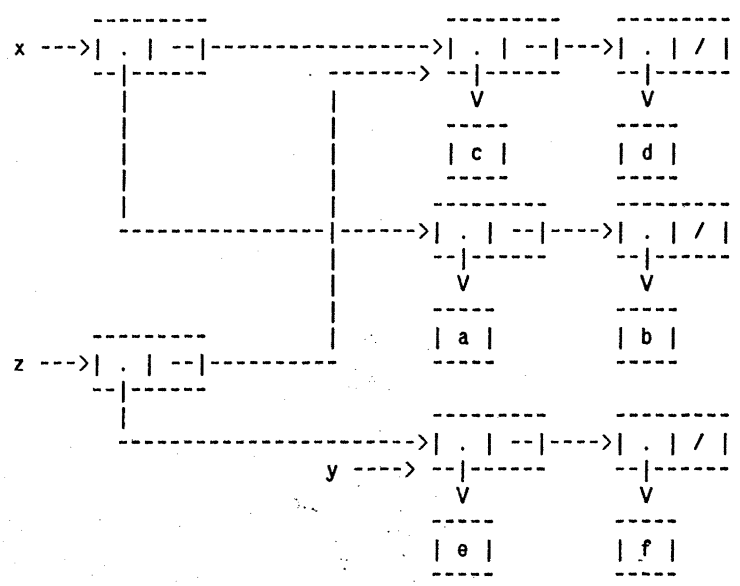


Figure 3-14: Effect of *(define z (cons y (cdr x)))* on the lists in figure 3-12.

The *set-cdr!* operation is similar to *set-car!*. The only difference is that the *cdr*

pointer of the pair, rather than the *car* pointer, is replaced. The effect of executing *(set-cdr! x y)* on the lists of 3-12 is shown in figure 3-15. Here the *cdr* pointer of ***x*** has been replaced by the pointer to *(e f)*. Also, the list *(c d)*, which used to be the *cdr* of ***x***, is now detached from the structure.

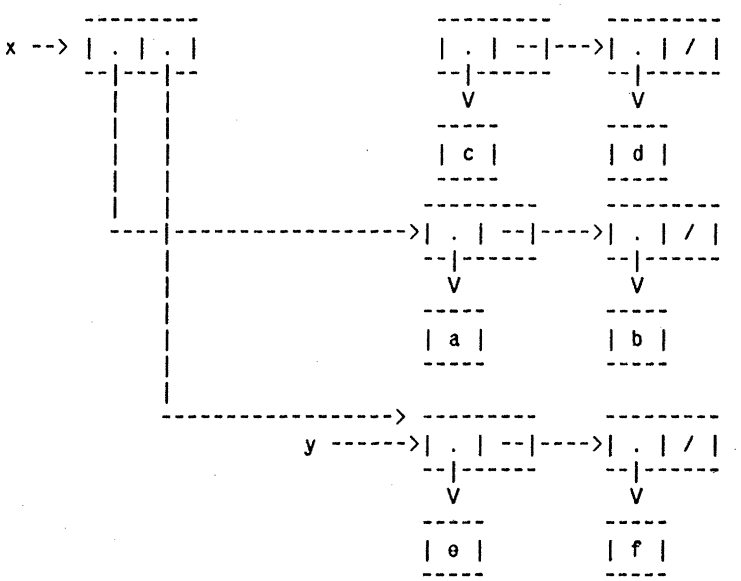


Figure 3-15: Effect of *(set-cdr! x y)* on the lists in figure 3-12.

Observe that *cons* builds new list structure by creating new pairs, whereas *set-car!* and *set-cdr!* modify existing pairs. Indeed, we could implement *cons* in terms of the two mutators, together with a procedure *get-new-pair*, which returns a new pair that is not part of any existing list structure. We obtain the new pair, set its *car* and *cdr* pointers to the designated objects, and return the new pair as the result of the *cons*.[14]

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

Exercise 3-11: The following procedure appends two lists:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

Append successively *cons*es the elements of *x* onto *y* to form a new list. The *append!* procedure is similar to *append* (if *x* is not *nil*), but it is a mutator rather than a constructor. It appends the lists by actually splicing them together, modifying the final pair of *x* so that its *cdr* is now *y*.

---

[14] *Get-new-pair* is one of the operations that must be implemented as part of the memory management required by a Lisp implementation. We will discuss this in Chapter 5, section 5.3.

```
(define (append! x y)
  (set-cdr! (last x) y)
  x)
```

Here `last` is a procedure that returns a pointer to the last pair in its argument:

```
(define (last x)
  (if (null? (cdr x))
      x
      (last (cdr x))))
```

Consider the interaction

```
==>(define x '(a b))
x
==>(define y '(c d))
y
==>(define z (append x y))
z
==>z
(a b c d)
==>(cdr x)
<exp1>
==>(define w (append! x y))
w
==>w
a b c d
==>(cdr x)
<exp2>
```

What are the expressions $\langle exp_1 \rangle$ and $\langle exp_2 \rangle$ printed by the interpreter? Draw box-and-pointer diagrams to explain your answer.

Exercise 3-12: Consider the following `make-cycle` procedure, which uses the `last` procedure defined in exercise 3-11:

```
(define (make-cycle x)
  (set-cdr! (last x) x)
  x)
```

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

```
(define z (make-cycle '(a b c)))
```

What happens if we try to compute `(last z)`?

Exercise 3-13: The following procedure is quite useful, although obscure:

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x nil))
```

Notice that `loop` uses the temporary variable `temp` to hold the old value of the `cdr` of `x`, since the `set-cdr!` on the next line destroys the `cdr`. Explain what `mystery` does in general. Suppose we execute the commands:

```
==>(define v '(a b c d))
==>(define w (mystery v))
```

What are `v` and `w` bound to now?

Sharing and identity

We already mentioned in section 3.1.2 the theoretical issues of "sameness" and "change" raised by introducing assignment. These issues arise in practice when individual pairs are *shared* among different data objects. For example, consider the structure formed by:

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

As shown in figure 3-16, *z1* is a pair whose *car* and *cdr* both point to the same pair *x*. This sharing of *x* by the *car* and *cdr* of *z* is a consequence of straightforward way in which *cons* is implemented. In general, using *cons* to construct lists will result in an interlinked structure of pairs, in which many individual pairs are shared by many different structures.

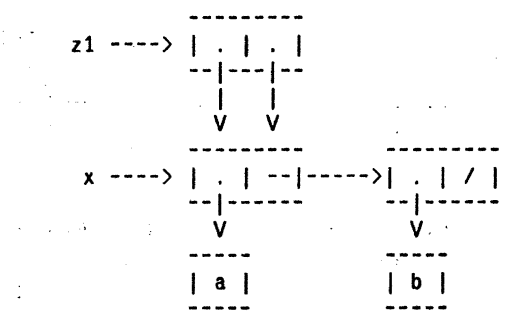


Figure 3-16: The list *z1* formed by *(cons x x)*.

In contrast to figure 3-16, figure 3-17 shows the structure created by

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

In this structure, the pairs in the two *(a b)* lists are distinct, although the actual atoms are shared.

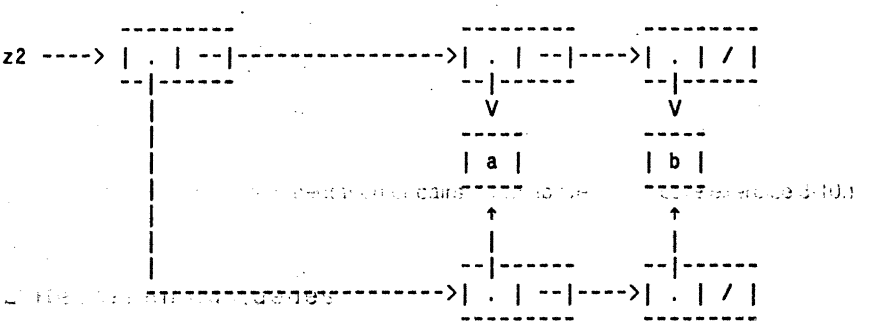


Figure 3-17: The list *z2* formed by *(cons (list 'a 'b) (list 'a 'b))*.

When interpreted as a list, *z1* and *z2* both represent "the same" list, *((a b) a b)*. In general, sharing is completely undetectable provided we operate on lists using only *cons*, *car*, and *cdr*. However, if we allow mutators on list structure, sharing becomes significant. As an example, consider the following procedure, which modifies the *car* of the structure to which it is applied.

```
(define (set-to-wow! x)
  (set-car! (car x) 'wow)
  x)
```

Even though *z1* and *z2* are "the same" tree structure, *set-to-wow!* applied to them yields different results. With *z1*, altering the *car* also changes the *cdr*, since in *z1*, the *car* and the *cdr* are the same pair. With *z2*, whose *car* and *cdr* are distinct, only the *car* is modified by *set-to-wow!*:

```
==>z1
((a b) a b)

==>(set-to-wow! z1)
((wow b) wow b)

==>z2
((a b) a b)

==>(set-to-wow! z2)
((wow b) a b)
```

One way to detect sharing in list structures is to use the predicate *eq?*, which we introduced in section 2.2.3 as a way to test if two symbols are equal. More generally, *(eq? x y)* tests whether *x* and *y* point to the same object. Thus, with *z1* and *z2* as defined in figures 3-16 and 3-17, we would have

```
(eq? (car z1) (cdr z1))
```

is true, while

```
(eq? (car z2) (cdr z2))
```

is false.

As we will see in the following sections, we can exploit sharing to greatly extend the repertoire of data structures that can be represented using pairs. On the other hand, sharing can also be dangerous, since modifications made to structures will also be "visible" to other structures that may share the modified parts. In general, the mutation operations *set-car!* and *set-cdr!* should be used with care; for, unless we have a good understanding of how our data objects are shared, mutation can have unanticipated results.[15]

Exercise 3-14: Draw box-and-pointer diagrams to explain the effect of *set-to-wow!* on the structures *z1* and *z2* above.

---

[15]The problems in dealing with sharing of mutable data objects reflect the underlying issues of "sameness" and "change" that we mentioned in section 3.1.2. As 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 precisely specified by giving its numerator and denominator. But this view is no longer valid once we allow mutation. For example, a bank account is still "the same" bank account, even if we change the balance by making a withdrawal; and conversely, we could have two different bank accounts with the same state information. Accordingly, a mutable compound data object has an "identity" that is something different from the pieces from which it is composed. In Lisp, we consider this "identity" to be the quality that is tested by *eq?*, i.e., by equality of pointers. Since in most Lisp implementations, a pointer is essentially a memory address, we are therefore "solving" the problem of identity 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 Lisp programs, but is hardly a general solution to the problem of "sameness" in computational models.

Exercise 3-15: Ben Bitdiddle decides to write a procedure 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 *car* plus the number in the *cdr* plus one more to count the current pair. And the number of pairs for an atom (including *nil*) is zero." So Ben writes the following procedure:

```
(define (count-pairs x)
  (if (atom? x)
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

Show that this procedure is not correct. In particular, draw box and pointer diagrams representing list structures made up of exactly 3 pairs for which Ben's procedure would return 3; return 4; return 7; never return at all.

Exercise 3-16: Devise a correct version of the *count-pairs* procedure of exercise 3-15 which will return the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure which is used to keep track of which pairs have already been counted.)

Write a procedure that examines a list and determines whether or not it contains a *cycle*, that is, it its ''tail'' points back within the list itself.

Redo exercise with an iterative process that uses only two state variables. (This requires a very clever idea.)

Mutation is just assignment

When we introduced compound data in Chapter 2, we showed in section 2.1.3 that we could construct compound data using only procedures as building blocks, as in the following implementation of pairs.

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined operation -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))
```

The same observation is true for mutable data. We can implement mutable data objects as procedures using assignment and local state. For instance, we can extend the above pair implementation to handle *set-car!* and *set-cdr!* in a manner analogous to the way we implemented bank accounts using *make-account* in section 3.1.1.

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Undefined operation -- CONS" m))))
  dispatch)

(define (car z) (z 'car))

(define (cdr z) (z 'cdr))

(define (set-car! z new-value)
  ((z 'set-car!) new-value)
  z)

(define (set-cdr! z new-value)
  ((z 'set-cdr!) new-value)
  z)
```

As we see, assignment is all that is needed, theoretically, to account for the behavior of mutable data, and to illustrate all of the problems inherent in mutability. On the other hand, from an implementation point of view, 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. In principle, we need only one means of changing the state of an object.

Exercise 3-17: Draw environment diagrams to illustrate the evaluation of the sequence

```
(define x (cons 1 2))
(define z (cons x x))
(set-car! (cdr z) 17)

==>(car x)
17
```

using the procedural implementation of pairs given above. (Compare exercise 3-10.)

3.3.2. Representing Queues

The mutators *set-car!* and *set-cdr!* enable us to use pairs to construct data structures that cannot be built using *cons*, *car* and *cdr* alone. In this section show how to use pairs to represent a data structure called a *queue*. In section 3.3.3 we will see how to represent *tables*.

A queue is a sequence in which items are inserted at one end, called the *rear* of the queue, and deleted at the other end, called the *front* the queue. 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. Notice that items are always removed in the order in which they are inserted. For this reason, a queue is sometimes called a *FIFO* (first in, first out) buffer.

Operation	Resulting queue
`(define 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 abstractions, we can regard a queue as defined by the following set of operations:

a constructor

- *make-queue* -- Takes no inputs and returns an empty queue, i.e., a queue containing no items.

two selectors

- *empty-queue?* -- Takes a queue as input. Returns *t* if the queue is empty and *nil* otherwise.
- *front* -- Takes a queue as input and returns the object at the front of the queue. Signals an error if the queue is empty. Does not modify the queue.

and two mutators

- *insert-queue!* -- Takes a queue and an object as inputs. Inserts the object in the queue (at the rear of the queue). Returns the modified queue as its value.
- *delete-queue!* -- Takes a queue as input and removes the item at the front of the queue. Returns the modified queue as its value.

Since a queue is a sequence of items, we could certainly represent it as an ordinary list: the *front* of the queue would be the *car* of the list, and inserting an item in the queue would amount to appending a new element at the end of the list. But 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 *cdr* operations, this scanning requires $O(n)$ time for a list of n items. Now we will see how making a simple modification to the list representation avoids this problem, allowing the queue operations to be implemented so that they require $O(1)$ time; that is, so that the time needed is independent of the length of the queue.

The problem with the list representation lies in the need to scan the list, and the reason we need to scan is that, although representing a queue as a list provides us with a pointer to the

front of the list, it gives us no easily accessible pointer to the rear. So the modification that solves the problem 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 the need to scan 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 considered a single object, we can use *cons* to combine the two pointers. Thus the queue itself will be the *cons* of the two pointers. Figure 3-19 illustrates this representation.

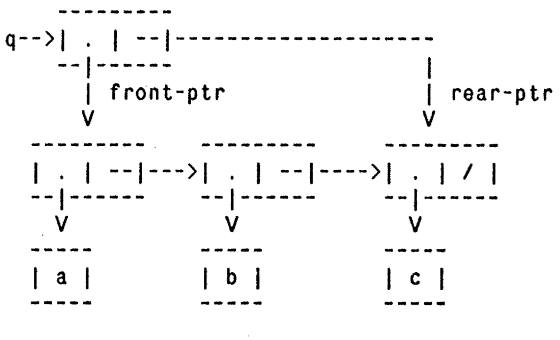


Figure 3-19: Implementation of a queue as a list with front and rear pointers.

To help us define the queue operations, we use the following procedures that enable us to select and to modify the front and rear pointers of a queue:

```
(define (front-ptr queue) (car queue))

(define (rear-ptr queue) (cdr queue))

(define (set-front-ptr! queue item) (set-car! queue item))

(define (set-rear-ptr! queue item) (set-cdr! queue item))
```

Now we can implement the actual operations. We will consider a queue to be empty if its front pointer is *nil*.

```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

The *make-queue* constructor returns, as an initially empty queue, a *cons* of two *nil*s.

```
(define (make-queue) (cons nil nil))
```

To select the item at the front of the queue, we return the *car* of the pair indicated by the front pointer:

```
(define (front queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (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 *car* is the item to be inserted and whose *cdr* is *nil*. 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.

```
(define (insert-queue! queue item)
 (let ((new-pair (cons item nil)))
   (cond ((empty-queue? queue)
          (set-front-ptr! queue new-pair)
          (set-rear-ptr! queue new-pair)
          queue)
         (else
          (set-cdr! (rear-ptr queue) new-pair)
          (set-rear-ptr! queue new-pair)
          queue))))
```

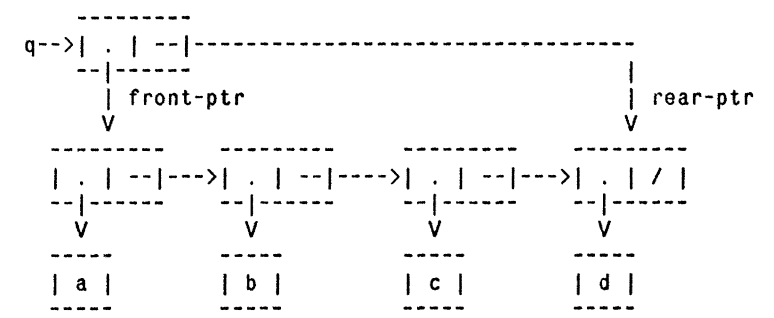


Figure 3-20: Result of *(insert-queue! q 'd)* on the queue of figure 3-19.

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 *cdr* pointer of the first item. Note that if the first item is the final item in the queue, the front pointer will be *nil* 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. If the queue is empty before the deletion, the procedure signals an error.

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "Delete called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

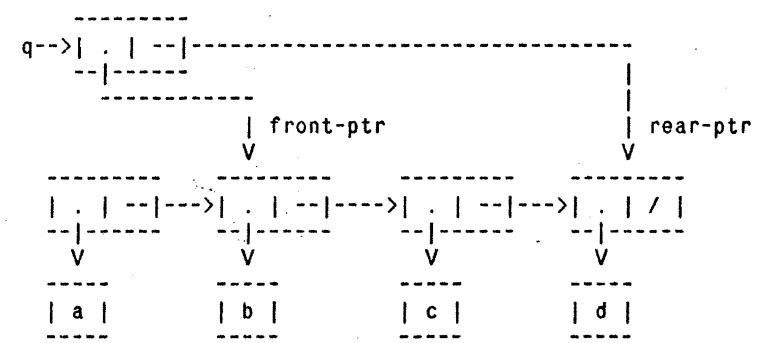


Figure 3-21: Result of *(delete-queue! q)* on the queue of figure 3-20.

Exercise 3-18: Ben Bitdiddle decides to test the queue implementation described above. He types in the procedures to the Lisp interpreter and proceeds to try them out:

```
==>(define q1 (make-queue))
q1

==>(insert-queue! q1 'a)
((a) a)

==>(insert-queue! q1 'b)
((a b) b)

==>(delete-queue! q1)
((b) b)

==>(delete-queue! q1)
(nil b)
```

"It's all wrong!" he complains. "The printout shows that the last item gets inserted into the queue twice. And when I deleted 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 the problem. "It's not that the items are going into the queue twice." she explains. "It's just that the standard Lisp 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 procedure for queues." Explain what Eva Lu is talking about. In particular, show why Ben's queue examples produce the printout that they do. Define a procedure *print-queue* that takes a queue as input and prints the sequence of items in the queue.

Exercise 3-19: Instead of representing a queue as a pair of pointers, we can build a queue as a procedure with local state. The local state will consist of pointers to the front and rear of an ordinary list. Thus the *make-queue* procedure will have the form

```
(define (make-queue)
  (let ((front-ptr ... )
        (rear-ptr ... ))
    <definitions of internal procedures>
    (define (dispatch m) ...)
    dispatch))
```

Complete the definition of *make-queue* and provide implementations of the queue operations using this representation.

Exercise 3-20: A *deque* ("double-ended queue") is a data structure consisting of a sequence in which items can be inserted and deleted either at the front or the rear of the sequence. The data access operations are *make-deque*, *empty-deque?*, *front-deque*, *rear-deque*, *front-insert-deque!*, *rear-insert-deque!*, *front-delete-deque!*, and *rear-delete-deque!*. Show how to represent deques using pairs, and give implementations of the operations. All operations should be accomplished in *O(1)* time.

3.3.3. Representing Tables

When we studied various ways of representing sets in Chapter 2, we mentioned in section 2.2.5 the problem of maintaining a table of records indexed by identifying *keys*. In the implementation of data-directed programming in section 2.3.3, we made extensive use of *two-dimensional tables*, in which information is stored and retrieved using a pair of keys. Here we see how to build tables as mutable list structures.

We first consider *one-dimensional tables*, in which each value is stored under a single key. We build the table as a list of pairs *<key, value>*, with an additional pair added at the beginning of the list. Adding the extra pair, a technique known as building a *headed list*, is a

useful trick in dealing with mutable lists, since it provides us with a fixed pair to change when we want to add something at the beginning of the list. Figure 3-22 shows the box-and-pointer diagram for the table

<a, 1> <b, 2> <c, 3>

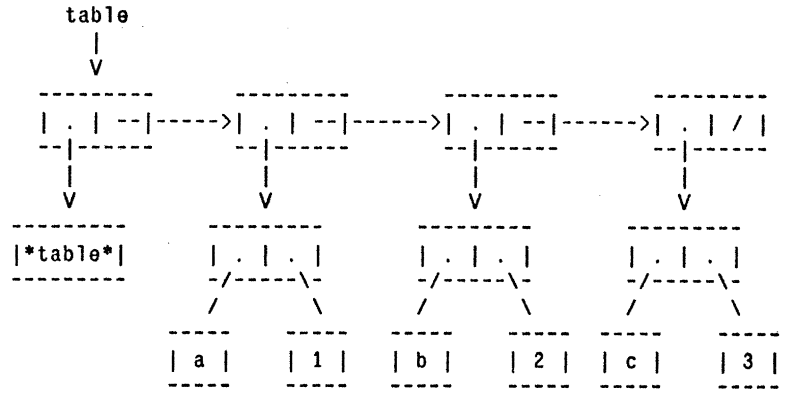


Figure 3-22: A table represented as a list of pairs.

To extract information from a table, we use the *lookup* procedure, which takes a key as argument and returns the associated value, or *nil* if there is no value stored under that key. *Lookup* is defined in terms of the *assq* operation, which expects a key and a list of pairs as arguments. *Assq* returns the pair that has the given *key* as its *car*. *Lookup* then checks to see that the pair is not null, then returns the *cdr* of the pair, which, as shown in figure 3-22, is the value that is stored under the key.

```
(define (lookup key table)
  (let ((pair (assq key (cdr table))))
    (if (null? pair)
        nil
        (cdr pair))))

(define (assq key pairs)
  (cond ((null? pairs) nil)
        ((eq? key (caar pairs)) (car pairs))
        (else (assq key (cdr pairs)))))
```

To insert a value in a table under a specified key, we first use *assq* to see if there is already a pair in the table with this key. If not, we form a new pair by *cons*ing the key with the value, and insert this at the head of the table's list of pairs. If there already is a pair with this key, we set the *cdr* of this pair to the designated new value. Note how the header of the table provides us with a fixed item to modify in order to insert the new pair. Thus, the same pair always starts the table, and *insert!* needn't return a new value for the start of the table when it adds a new pair.

```
(define (insert! key value table)
  (let ((pair (assq key (cdr table))))
    (if (null? pair)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))
        (set-cdr! pair value))))
```

To construct a new table, we simply create a header pair, with the (arbitrarily chosen) symbol **table** as its *car*.

```
(define (make-table)
  (cons '*table* nil))
```

Two-dimensional tables

In a two-dimensional table, each value is indexed by a pair of keys. We can construct such a table as a one-dimensional table in which each key identifies a subtable. 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 pair within the subtable. (Note that the pair containing the first key and the subtable serves as the header of the subtable.)

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assq key-1 (cdr table))))
    (if (null? subtable)
        nil
        (let ((pair (assq key-2 (cdr subtable))))
          (if (null? pair)
              nil
              (cdr pair))))))
```

To insert a new item under a pair of keys, we use *assq* to see if there is a subtable stored under the first key. If not, we build a new subtable containing the single pair *<key-2, value>* and add it to the table. If a subtable already exists for the first key, we add the new pair to this subtable, using the insertion method for one-dimensional tables described above.

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assq key-1 (cdr table))))
    (if (null? subtable)
        (set-cdr! table
                  (cons (cons key-1
                              (cons (cons key-2 value) nil))
                        (cdr table)))
        (let ((pair (assq key-2 (cdr subtable))))
          (if (null? pair)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))
              (set-cdr! pair value))))))
```

Creating local tables

The *lookup* and *insert!* operations defined above take the table as an argument, which allows us to use systems that have more than one table. Another way to deal with multiple tables is to have separate *lookup* and *insert!* procedures for each table. We can do this by representing the table procedurally as an object that maintains an internal table as a local state variable, and which, when asked, will supply the appropriate procedures with which to operate on the table. Here is a generator for two-dimensional tables implemented in this

fashion.

```
(define (make-table)
  (let ((local-table (cons '*table* nil)))

    (define (lookup key-1 key-2)
      (let ((subtable (assq key-1 (cdr local-table))))
        (if (null? subtable)
            nil
            (let ((pair (assq key-2 (cdr subtable))))
              (if (null? pair)
                  nil
                  (cdr pair))))))

    (define (insert! key-1 key-2 value)
      (let ((subtable (assq key-1 (cdr local-table))))
        (if (null? subtable)
            (set-cdr! local-table
                      (cons (cons key-1
                                  (cons (cons key-2 value) nil))
                            (cdr local-table)))
            (let ((pair (assq key-2 (cdr subtable))))
              (if (null? pair)
                  (set-cdr! subtable
                            (cons (cons key-2 value)
                                  (cdr subtable)))
                  (set-cdr! pair value))))))

    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation -- TABLE" m))))

    dispatch))
```

For example, we could implement the *get* and *put* operations used in section 2.3.3 for data-directed programming, as follows:

```
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```

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 3-21: In the table implementations above, the keys are tested for equality using *eq?*. This is not always the appropriate test. For instance, with numerical keys, we should test equality using =. (Whether or not two instances of the same number are *eq?* (i.e., represented by equal pointers) is highly implementation dependent.) Design a table constructor *make-table* that takes as an argument a *same-key?* procedure that will be used to test equality of keys. *Make-table* should return a *dispatch* procedure that can be used to access appropriate *lookup* and *insert!* procedures for a local table.

Exercise 3-22: Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys, and where different values may be stored using different numbers of keys. The *lookup* and *insert!* procedures should take as input a list of keys used to access the table.

Exercise 3-23: In order to search a table as implemented above, one needs to scan through the list of keys. This is basically the *unordered list representation* of section 2.2.5. For large tables, it may be more

efficient to structure the table in a different manner. Describe a table implementation where the *(<key>, <value>)* pairs are structured using a binary tree, assuming that keys can be ordered in some way (e.g., numerically or alphabetically). (Compare exercise 2-36 of Chapter 2.)

Exercise 3-24: *Memoization* (also called *tabulation*) is a technique that enables a procedure to record, in a local table, various values which have previously been computed. Using this technique can make a vast difference in the performance of a program. A memoized procedure maintains a one-dimensional table in which values of the function are stored using, as keys, the arguments that produced the values. When the memo-function is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table.

As an example of memoization recall from section 1.2.2 the exponential process for computing Fibonacci numbers:

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

The memoized version of the same procedure is

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

where the memoizer is defined as:

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((seen (lookup x table)))
        (if (not (null? seen))
            seen
            (let ((ans (f x)))
              (insert! x ans table)
              ans))))))
```

Draw an environment diagram to analyze the computation of *(memo-fib 3)*. Explain why *memo-fib* computes the *n*th Fibonacci number in time proportional to *n*. Why must the internally defined *fib* procedure call *memo-fib* rather than *fib* itself in order for this scheme to work? (To run *memo-fib*, we should be sure that the table operations are constructed so as to test keys for numeric equality. See exercise 3-21 above.)

3.3.4. A Simulator for Digital Circuits

The design of complex digital systems, such as computers, is an important engineering problem. 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.

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 which depends on the type of the function box.

For example, an *inverter* is a primitive function box that inverts the sense of 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 3-23.

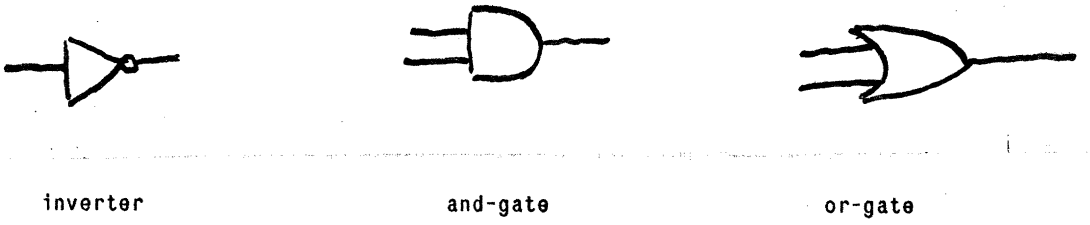


Figure 3-23: Primitive functions in the digital logic simulator.

An *and-gate*, also shown in figure 3-23, is a primitive function box with two inputs and one output. It drives its output signal to a value which is the *logical and* of the inputs. That is, if both of its input signals are 1 then, one and-gate delay time later, the and-gate will force its output signal to be a 1, and otherwise to be a 0. An *or-gate* is a similar two-input primitive function box, which drives its output signal to a value which is the *logical or* of the inputs. That is, the output will become 1 if at least one of the input signals is 1, and otherwise the output will become 0.

We can connect primitive functions together to construct more complex functions. This is accomplished by wiring the outputs of some function boxes to the inputs of other function boxes. For example, a *half-adder circuit*, shown in figure 3-24, is a circuit consisting 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, due to 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.

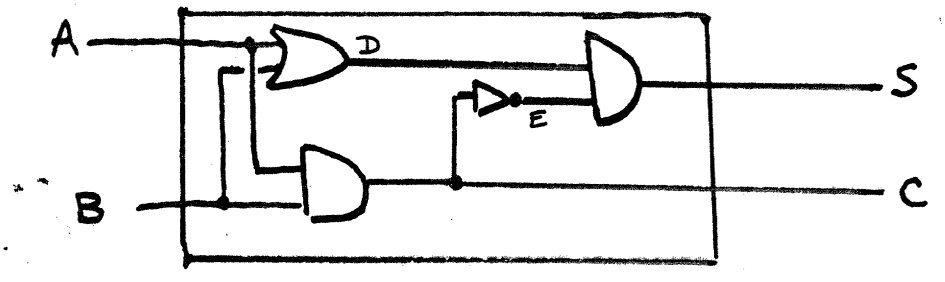


Figure 3-24: A half-adder circuit.

We now build a computational model corresponding to the conceptual structure of the digital logic circuits we wish to study. We will have computational objects modeling the wires,

which will "hold" the signals. The function boxes will be modeled by procedures that enforce the correct relationships among the signals.

One basic element of our simulation will be a procedure *make-wire*, which constructs new wires. For example, we can construct six wires as follows:

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

We attach a function box to a set of wires by calling a procedure for making that kind of box, with the wires to be attached as its arguments. For example, given that we can construct *and-gates*, *or-gates*, and *inverters*, we can wire up the half-adder shown in figure 3-24:

```
(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 to be a *half-adder* by defining a procedure that constructs this circuit, given the four external wires to be attached to the half-adder:

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)))
```

The importance of this way of proceeding is that we can now use *half-adder* itself as a building block in creating more complex circuits. Figure 3-25, for example, shows a *full-adder*, which is composed of two half-adders and an or-gate.[16] We can construct a full-adder as

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)))
```

Having defined *full-adder* as a procedure, we can now use it in turn as a building block for creating still more complex circuits. (For example, see exercise 3-27.)

---

[16] A full-adder is a basic circuit element used in adding two binary numbers. *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.

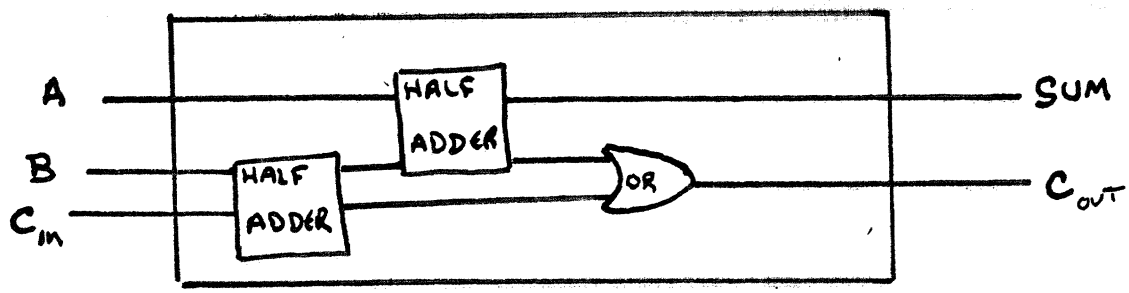


Figure 3-25: A full-adder circuit.

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 Lisp in section 1.1, we can say that the primitive function boxes form the *primitive elements* of the language, that wiring boxes together provides a *means of combination*, and that specifying wiring patterns as procedures 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 these, 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> <procedure of no arguments>)
: Specifies that the designated procedure should be run whenever the signal on the wire changes value. Such procedures 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 procedure, `after-delay`, which takes a time delay and a procedure to be run, and which executes the given procedure after the given delay.

Using these procedures, we can define the primitive logic functions. To connect an `input` to an `output` through an `inverter`, we use `add-action!` to associate with the input wire a procedure that will be run whenever the signal on the input wire changes value. The procedure computes the `logical-not` of the input signal, and then, after one `inverter-delay`, sets the output signal to be this new value:[17]

---

[17] We also need a procedure `logical-not`:

```
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))
```

together with analogous procedures `logical-and` and `logical-or`.

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output
                                  new-value)))))
  (add-action! input invert-input))
```

An *and-gate* is more complex. The action procedure must be run if either input changes. It computes the *logical-and* 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*.

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure))
```

Exercise 3-25: Define an *or-gate* as a primitive function box. Your definition should be similar to the one for *and-gate*.

Exercise 3-26: Another way to construct an or-gate is as a compound digital logic device, built from and-gates and inverters. Define a procedure *or-gate* which accomplishes this. What is the delay time of the or-gate in terms of *and-gate-delay* and *inverter-delay*?

Exercise 3-27: Figure 3-26 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 $a_1a_2a_3\ldots a_n$ and $b_1b_2b_3\ldots b_n$ are the two binary numbers to be added (each a_k and b_k is a 0 or a 1). The circuit generates $s_1s_2s_3\ldots s_n$, the n bits of the sum, and c, the carry from the addition. Write a procedure *ripple-carry-adder* that generates this circuit. The procedure 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 that it is slow, due to 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?

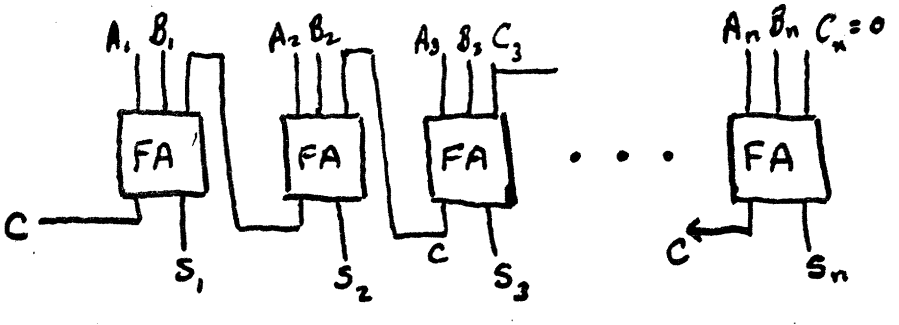


Figure 3-26: A ripple-carry adder for n-bit numbers.

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-procedures* to be run when the signal changes value. We implement the wire as a collection of local procedures, together with a *dispatch* procedure that will select the appropriate local operation, just as we did with the simple bank account object in section 3.1.1.

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures nil))
    (define (set-my-signal! new-value)
      (if (not (eq? signal-value new-value))
          (sequence (set! signal-value new-value)
                    (call-each action-procedures))
          'done))

    (define (accept-action-procedure proc)
      (set! action-procedures
        (cons proc action-procedures))
      (proc))

    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure)
            (else (error "Unknown operation -- WIRE" m))))

    dispatch))
```

The local procedure *set-my-signal!* sees if the new signal value changes the signal on the wire. If so, it runs each of the action procedures, using the following procedure, which calls each of the procedures in a list:

```
(define (call-each procedures)
  (if (null? procedures)
      nil
      (sequence
       ((car procedures))
       (call-each (cdr procedures)))))
```

The local procedure *accept-action-procedure* adds the given procedure to the list of procedures to be run, and then runs the new procedure once. (See exercise 3-28.)

With the local *dispatch* procedure set up as specified, we can provide the following

procedures to access the local operations on wires:[18]

```
(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))

(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))
```

The agenda and driver loop

The only thing left to do to finish building our logic simulator is to construct *after-delay*. The idea here is that we maintain a data structure, called an *agenda*, which contains a schedule of things to do in the simulated future. The particular agenda that we are using is stored in the global variable *the-agenda*.

After-delay is a procedure that adds new elements to the agenda stored in *the-agenda* using a mutator on agendas, *add-to-agenda!*. The current simulation time is obtained using the *current-time* procedure.

```
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda))
```

The simulator is driven by the procedure *propagate* which operates on *the-agenda*, executing each procedure on the agenda in sequence. It presumes that *the-agenda* is a data structure with certain operations defined on it, specifically:

empty-agenda? True if the specified agenda is empty.

first-agenda-item
Returns the first item on an agenda.

remove-first-agenda-item!
Modifies the agenda by removing the first item.

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:

---

[18] These procedures are simply "syntactic sugar" that allow us to use ordinary procedural syntax to access the local procedures of objects. It may seem curious that we can interchange the role of "procedures" and "data" in such a simple way. For example, if we write

```
(wire 'get-signal)
```

we think of *wire* as a procedure 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 procedure *get-signal*. The truth of the matter is that, in a language where we can deal with procedures as objects, there is no fundamental difference between "procedures" and "data," and we can choose our syntactic sugar to allow us to program in whatever style we choose.

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

A sample simulation

To show the simulator in action, we use the following procedure, which places a "probe" on a wire. This tells the wire that, whenever its signal changes value, it should print the new signal value, together with the current time and an identifying name.

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (print name)
                 (princ (current-time the-agenda))
                 (princ "  New-value = ")
                 (princ (get-signal wire)))))
```

We begin by initializing the agenda and specifying delays for the primitive function boxes:

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

Now we define four wires, placing probes on two of them:

```
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define 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 shown in figure 3-24, 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
DONE
```

As shown, the *sum* signal changes to 1 at time 8. We are now 8 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
DONE
```

As shown, the *carry* changes to 1 at time 11, and the *sum* changes to 0 at time 16.

Exercise 3-28: The internal procedure *accept-action-procedure* used by *make-wire* specifies that whenever a new action procedure is added to a wire, the procedure is first 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-procedure* as

```
(define (accept-action-procedure proc)
  (set! action-procedures (cons proc action-procedures)))
```

Implementing the agenda

Finally, we give details of the *agenda* data structure, which holds the procedures that are scheduled for future execution. There are the following operations for manipulating agendas:

We can add a new action to an agenda to be considered at a (future) simulation time with *(add-to-agenda! <time> <action> <agenda>)*. We can get the current simulation time with *(current-time <agenda>)*. We can get the current action from the agenda with *(first-agenda-item <agenda>)*. We can delete the current action from the agenda with *(remove-first-agenda-item! <agenda>)*. And we can determine that we are finished with *(empty-agenda? <agenda>)*.

The agenda is made up of time segments. Each time segment is a pair consisting of a number (the time) and an associated queue that holds the procedures that are scheduled to be run during that time segment.

```
(define (make-time-segment time queue)
  (cons time queue))

(define (segment-time s) (car s))

(define (segment-queue s) (cdr 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. We construct a new agenda as a headed list with an initial empty time segment at the initial 0 time.

```
(define (make-agenda)
  (cons '*agenda*
        (cons (make-time-segment 0 (make-queue))
              nil)))
```

The current time will always be the time associated with the first time segment on the agenda.

```
(define (segments agenda) (cdr agenda))
```

```
(define (first-segment agenda) (car (segments agenda)))

(define (rest-segments agenda) (cdr (segments agenda)))

(define (set-segments! agenda segments)
  (set-cdr! agenda segments))

(define (current-time agenda)
  (segment-time (first-segment agenda)))
```

An agenda is empty when the first segment has an empty queue and there are no more segments to be executed:

```
(define (empty-agenda? agenda)
  (and (empty-queue? (segment-queue (first-segment agenda)))
       (null? (rest-segments agenda))))
```

To add an item to the agenda, we scan the agenda looking at the time of each segment. If we find a segment for our appointed time, we add our action to the associated queue. If we hit the end of the agenda, we must create a new time segment at the end. If we hit a time later than the one we are appointed to we must must insert a new time segment into the agenda just before it. Otherwise we continue scanning. Note that the appointed time can never be earlier than the time of the first agenda segment (the current time).

```
(define (add-to-agenda! time action agenda)
  (if (= (segment-time (first-segment agenda)) time)
      (insert-queue! (segment-queue (first-segment agenda))
                     action)
      (cond ((null? (rest-segments agenda))
             (insert-new-time! time action (segments agenda)))
            ((> (segment-time (car (rest-segments agenda))
                              time)
                time)
             (insert-new-time! time
                               action
                               (segments agenda)))
            (else
             (add-to-agenda! time
                             action
                             (segments agenda))))))
```

Inserting a new time segment is accomplished by editing the top-level structure of the agenda.

```
(define (insert-new-time! time action segments)
  (let ((q (make-queue)))
    (insert-queue! q action)
    (set-cdr! segments
              (cons (make-time-segment time q)
                    (cdr segments)))))
```

The procedure that removes the first item from the agenda assumes the first item will always be at the front of the queue in the first segment of the agenda.

```
(define (remove-first-agenda-item! agenda)
  (delete-queue! (segment-queue (first-segment agenda))))
```

The fact that the first agenda item must always be in the first segment of the agenda is arranged by the procedure that finds a new first item. The implementation assumes that this procedure will not be called if the agenda is empty.

```
(define (first-agenda-item agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (if (empty-queue? q)
        (sequence (set-segments! agenda (rest-segments agenda)))
                  (first-agenda-item agenda))
        (front q))))
```

3.3.5. Propagation of Constraints

Computer programs are traditionally organized in terms of *one-directional* computations: They perform operations on pre-specified 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

$$d\,A\,E = F\,L$$

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 procedure 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.[19]

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 express that certain relations must hold between quantities. For example,

```
(adder a b c)
```

specifies that the quantities a, b, and c must be related by the equation $a+b=c$. Other primitive constraints are

```
(multiplier x y z)
```

which expresses the constraint $xy = z$, and

```
(constant <number> x)
```

which says that the value of x must be equal to the designated number.

Our language provides a *means of combining* primitive constraints in order to express more

---

[19] This idea first appeared in the incredibly forward-looking SKETCHPAD system by Ivan Sutherland (1963!) [47]. A beautiful constraint propagation system was developed by Alan Borning [3] at Xerox Palo Alto Research Center based on the Smalltalk language. Sussman, Stallman, and Steele applied this idea to electrical circuit analysis [45, 46]. TK!Solver, an excellent commercial system of this sort developed by Software Arts, Inc., became available for use on personal computers in 1983.

complex relations. We combine constraints by constructing *constraint networks*, in which constraints are joined via *connectors*. A connector is an object which "holds" a value that may participate in one or more constraints. For example, we know that the relationship between Fahrenheit and Centigrade temperatures is

```
9 C = 5 (F - 32)
```

Such a constraint can be thought of as a network, shown in figure 3-27, consisting of primitive adder, multiplier, and constant constraints.

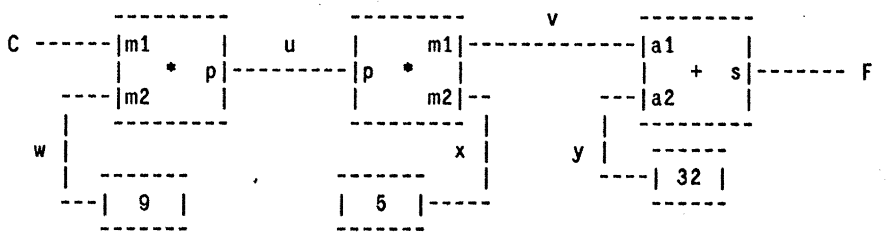


Figure 3-27: The relation *9 C = 5 (F - 32)* expressed as a constraint network.

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 Centigrade temperature. The *m2* terminal is linked to a connector *w*, which is also connected 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.

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 all of its associated constraints, and so on. For instance, in the Centigrade/Fahrenheit conversion, *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*9 = 225. *U* then awakens the second multiplier, which then sets *v* to 45. *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:

```
(define C (make-connector))
(define F (make-connector))
(centigrade-fahrenheit-converter C F)
```

The procedure that creates the network is defined as follows:

```
(define (centigrade-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)))
```

This procedure creates the internal connectors *u*, *v*, *w*, *x*, *y*, and links them as shown in figure 3-27, using the primitive constraint boxes `adder`, `multiplier`, and `constant`. Observe that, just as with the digital circuit simulator of section 3.3.4, expressing these combinations of primitive elements in terms of procedures 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` procedure 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 "Centigrade temp" C)
(probe "Fahrenheit temp" F)
```

Next we set the value of *C* to 25. (The extra argument tells *C* that this directive comes from the `user`.)

```
==>(set-value! C 25 'user)
Probe: Centigrade temp = 25
Probe: Fahrenheit temp = 77
done
```

The probe on *C* awakens and reports the value. *C* also propagates its value through the network as described above, which sets *F* to 77, as we can see from 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:  Centigrade temp = ?
Probe:  Fahrenheit temp = ?
done
```

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. So *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:  Centigrade temp = 100
done
```

This new value, when propagated through the network, entails that *C* must 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 also *F* given *C*. Indeed, this "non-directionality of computation" is the distinguishing feature of constraint-based systems.

Implementing the constraint system

The constraint system is implemented using procedural objects with local state in a manner very similar to the digital circuit simulator of section 3.3.4. While 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 as follows:

(has-value? <connector>)
: tells whether the connector currently has a value.

(get-value <connector>)
: returns the connector's current value.

(set-value! <connector> <new-value> <informant>)
: tells the connector that some informant is requesting it to set its value to a new value.

(forget-value! <connector> <retractor>)
: tells the connector that some retractor is requesting it to forget its value.

(connect <connector> <new-constraint>)
: tells the connector that it should participate in a new constraint.

Connectors communicate with constraints using the procedures *inform-about-value*, which tells the designated constraint that the connector has a value, and *inform-about-no-value*, which tells the constraint that the connector has lost its value.

Here is the primitive *adder* constraint that is established between summands *a1* and *a2*, and a *sum* connector. The *adder* itself is implemented as a procedure with local state.

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2))
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
                       (- (get-value sum) (get-value a1))
                       me))
          ((and (has-value? a2) (has-value? sum))
           (set-value! a1
                       (- (get-value sum) (get-value a2))
                       me))))

  (define (process-forget-value)
    (forget-value! sum me)
    (forget-value! a1 me)
    (forget-value! a2 me)
    (process-new-value))

  (define (me request)
    (cond ((eq? request 'I-have-a-value)  process-new-value)
          ((eq? request 'I-lost-my-value) process-forget-value)
          (else (error "Unknown request -- ADDER" request))))

  (connect a1 me)
  (connect a2 me)
  (connect sum me)
  me)
```

The adder's local procedure *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. Notice that 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. Then it runs *process-new-value*. The reason for this last step is that one or more connectors may still have a value (e.g., the 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*.

The adder's internal *me* procedure acts as a dispatch to the local procedures. The following "syntax interfaces" (see footnote in section 3.3.4) are used in conjunction with the dispatch:

```
(define (inform-about-value constraint)
  ((constraint 'I-have-a-value)))

(define (inform-about-no-value constraint)
  ((constraint 'I-lost-my-value)))
```

Finally, the *adder* connects itself to the designated connectors, and returns as its value the dispatch procedure *me*.

A *multiplier* is very similar to an *adder*. Notice that it will set its *product* to 0 if either of the factors is 0, even if the other factor is not known.

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                       (* (get-value m1) (get-value m2))
                       me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2
                       (/ (get-value product) (get-value m1))
                       me))
          ((and (has-value? product) (has-value? m2))
           (set-value! m1
                       (/ (get-value product) (get-value m2))
                       me))))

  (define (process-forget-value)
    (forget-value! product me)
    (forget-value! m1 me)
    (forget-value! m2 me)
    (process-new-value))

  (define (me request)
    (cond ((eq? request 'I-have-a-value)  process-new-value)
          ((eq? request 'I-lost-my-value) process-forget-value)
          (else (error "Unknown request -- MULTIPLIER" request))))

  (connect m1 me)
  (connect m2 me)
  (connect product me)
  me)
```

A *constant* constraint 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.

```
(define (constant value connector)
  (define (me request)
    (error "Unknown request -- constant" request))

  (connect connector me)
  (set-value! connector value me)
  me)
```

Finally, a *probe* prints a message about the setting or unsetting of the designated connector:

```
(define (probe name connector)
  (define (process-new-value)
    (print "Probe: ")
    (princ name)
    (princ " = ")
    (princ (get-value connector)))

  (define (process-forget-value)
    (print "Probe: ")
    (princ name)
    (princ " = ")
    (princ "?"))
```

```
(define (me request)
  (cond ((eq? request 'I-have-a-value)  process-new-value)
        ((eq? request 'I-lost-my-value) process-forget-value)
        (else (error "Unknown request -- PROBE" request))))

(connect connector me)
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.

```
(define (make-connector)
  (let ((value nil) (informant nil) (constraints nil))
    (define (set-my-value newval setter)
      (cond ((not (has-value? me))
             (set! value newval)
             (set! informant setter)
             (for-each-except setter
                              inform-about-value
                              constraints))
            ((not (= value newval))
             (error "Contradiction" (list value newval)))))

    (define (forget-my-value retractor)
      (if (eq? retractor informant)
          (sequence (set! informant nil)
                    (for-each-except retractor
                                     inform-about-no-value
                                     constraints))))

    (define (connect new-constraint)
      (if (not (memq new-constraint constraints))
          (set! constraints (cons new-constraint constraints)))
      (if (has-value? me)
          (inform-about-value new-constraint)))

    (define (me request)
      (cond ((eq? request 'has-value?) (not (null? informant)))
            ((eq? request 'value) value)
            ((eq? request 'set-value!) set-my-value)
            ((eq? request 'forget) forget-my-value)
            ((eq? request 'connect) connect)
            (else (error "Unknown operation -- CONNECTOR" request))))

    me))
```

The connector's local procedure *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. Then the connector will notify all of its participating constraints, except for the constraint that requested the *set*. This is accomplished using the following iterator, which applies a designated procedure to all items in a list, except for a given one:

```
(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items)))))
  (loop list))
```

If a connector is asked to forget its value, it runs the local procedure *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 procedure *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 procedure *me* serves as a dispatch to the other internal procedures, and also represents the connector as an object. The following procedures provide a syntax interface for the dispatch.

```
(define (has-value? connector)
  (connector 'has-value?))

(define (get-value connector)
  (connector 'value))

(define (forget-value! connector retractor)
  ((connector 'forget) retractor))

(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))

(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

Exercise 3-29: Using primitive *multiplier*, *adder* and *constant* constraints, define a procedure *averages*, 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 3-30: 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*.

```
(define (squarer a b)
  (multiplier a a b))
```

Louis realizes that his *squarer* will not automatically complain if *b* goes negative. But there is a much more serious flaw in his idea. Explain.

Exercise 3-31: Ben Bitdiddle tells Louis that one way around the problem in exercise 3-30 is to define *squarer* as a new primitive constraint. Fill in the missing portions in Ben's outline for a procedure to implement such a constraint:

```
(define (squarer a b)
  (define (process-new-value)
    (cond ((and (has-value? b) (< (get-value b) 0))
           (error "square less than 0 -- SQUARER" (get-value b)))
          <rest of clauses>))

  (define (process-forget-value) ...)
  (define (me request) ...)
  <rest of definition>
  me)
```

Exercise 3-32: Suppose we execute the following sequence of commands in the global environment.

```
(define a (make-connector))
(define b (make-connector))
(define c (make-connector))
(set-value! a 10 'user)
```

At some time during the execution of the last command, the following expression from the connector's local procedure 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 3-33: The *centigrade-fahrenheit-converter* procedure is cumbersome when compared with a more expression-oriented style of definition, such as

```
(define (centigrade-fahrenheit-converter x)
  (c+ (c* (c/ (cv 9) (cv 5))
          x)
      (cv 32)))

(define C (make-connector))
(define F (centigrade-fahrenheit-converter C))
```

Here *c+*, *c**, etc. are the "constraint" versions of the arithmetic operations. For example, *c+* takes two connectors as arguments and returns a connector that is related to these by an *adder* constraint:

```
(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))
```

Define analogous procedures *c-*, *c**, *c/* and *cv* (constant value) that enable us to define compound constraints as in the converter example above.[20]

---

[20] The reason that the expression-oriented format is more convenient is that it avoids the need to explicitly name the intermediate expressions in the 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 want to compute *(a+b)*(c+d)* where the variables represent vectors, we could work in "imperative style" using procedures that set the values of designated vector arguments, but which do not themselves return vectors as values:

```
(v-sum a b temp1)
(v-sum c d temp1)
(v-prod temp1 temp2 answer)
```

Alternatively, we could could work in terms of expressions, using procedures that return vectors as values, thus avoiding the need to explicitly mention *temp1* and *temp2*:

```
(define answer (v-prod (v-sum a b) (v-sum c d)))
```

Since Lisp allows us to return compound objects as values of procedures, 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.

3.4. Stream Processing

This section introduces new compound data structures called *streams*. We use streams to organize computations on collections of data in a way that corresponds in spirit to an electrical engineer's concept of a signal processing system. Organizing computations in this way greatly enhances our ability to formulate abstractions that capture common patterns of data manipulation. Indeed, we will see how a few elegant stream operations can succinctly express the structural similarity of a wide range of programs. From an abstract point of view, a stream is simply a sequence of data objects. However, we will find that the straightforward implementation of streams as lists does not allow us to fully exploit the power of stream processing. To solve this problem, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) data structures as streams.

In the previous sections of this chapter we used assignment and local state to model objects and change. In this section, we will see how streams form the basis for a very different approach to modeling. Instead of using objects with changing local state, we construct a stream that represents the "time history" of the system being modeled. A consequence of this strategy is that it allows us to model systems that have state, without ever using assignment or mutable data. This has important implications, both theoretical and practical, for it enables us to build models that avoid the problems inherent in introducing assignment that we discussed in section 3.1.2. On the other hand, the stream framework raises problems of its own, and the question of which modeling technique leads to more modular and easily maintained systems remains open.

3.4.1. Streams as Standard Interfaces

In Chapter 1, section 1.3, we saw how program abstractions, implemented as higher order procedures, can capture common patterns of usage in programs that deal with numerical data. We would now like to formulate analogous operations for working with compound data. Unfortunately, the style in which we have been writing procedures often masks the commonality that underlies many typical computations. Consider, for example, a procedure that takes as argument a binary tree, all of whose leaves are integers, and computes the sum of the squares of the leaves that are odd.

```
(define (sum-odd-squares tree)
  (if (leaf-node? tree)
      (if (odd? tree)
          (square tree)
          0)
      (+ (sum-odd-squares (left-branch tree))
         (sum-odd-squares (right-branch tree)))))
```

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

```
(define (odd-fibs n)
  (define (next k)
    (if (> k n)
        '()
        (let ((f (fib k)))
          (if (odd? f)
              (cons f (next (1+ k)))
              (next (1+ k))))))
  (next 1))
```

Despite the fact that these two procedures are structurally very different, a more abstract description of the two computations reveals a great deal of similarity:

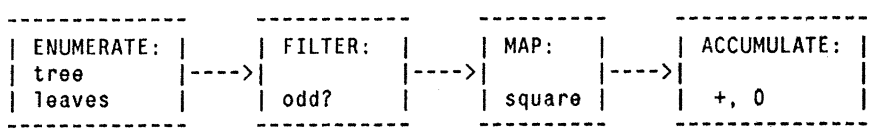
The first program

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

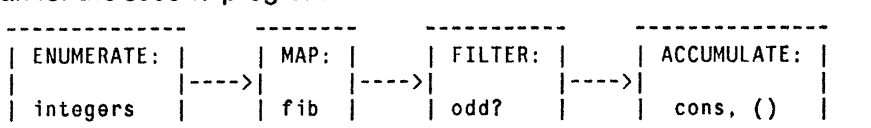
The second program

- Enumerates the integers from 1 to *n*;
- computes the Fibonacci number for each integer;
- filters them, selecting the odd ones;
- accumulates the results into a list, using *cons*, starting with the empty list.

An electrical engineer would find it easy to conceptualize these processes in terms of signals flowing through a cascade of stages, each of which implements part of the program plan. The first program realizes the following signal flow plan:



We begin with an *enumerator*, which generates a "signal" consisting of the leaves of a given tree. This signal is passed through a *filter*, which eliminates all but the odd elements. The resulting signal is in turn passed through a *map*, which is a "transducer" that applies the *square* procedure to each element. The output of the map is then fed to an *accumulator*, which combines the elements using +, starting from an initial 0. Here is an analogous signal flow plan for the second program:



Unfortunately, the two procedures above fail to exhibit this signal flow structure. For instance, if we examine the *sum-odd-squares* procedure, we find that the *enumerate* is implemented partly by the *leaf-node?* test and partly by the tree-recursive structure of the

procedure. Similarly, the *accumulate* is found partly in the test and partly in the addition used in the recursion. In general, there are no distinct parts of either procedure that correspond to the elements in our signal flow description. The procedures decompose the computations in a different way, spreading the enumeration over the program, mingling it with the map, the filter, and the accumulation. If we could organize our programs so that the signal flow structure were manifest in the procedures we write, this would increase the conceptual clarity of the resulting code. It would also provide identifiable enumerate, filter, map, and accumulate program elements that we could combine in a "mix and match" way to construct programs from standard, well-understood pieces.

Stream operations

Our traditional program organization concentrates on the order of events in a computation, rather than on the flow of data. Thus 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. We will implement these signals as data structures called *streams*, and we observe from our signal flow diagrams that a stream is simply a sequence of elements. We can define streams abstractly, in terms of a constructor *cons-stream* and two selectors *head* and *tail*. These are related by the following condition:

- For any objects *a* and *b*, if *x* is *(cons-stream a b)* then *(head x)* is *a* and *(tail x)* is *b*.

We will also assume that there is an object called *the-empty-stream*, which contains no elements, and a predicate *empty-stream?*, which tests whether a given stream is empty.

Notice that as far as this data abstraction is concerned, ordinary Lisp pairs provide a perfectly adequate implementation for streams: *cons-stream*, *head*, and *tail* can be implemented as *cons*, *car*, and *cdr*, respectively, *the-empty-stream* can be the empty list, and *empty-stream?* can be the predicate *null?*. Indeed, the above condition that defines the relationship among the three stream operations is identical to the condition that we used to define *cons*, *car*, and *cdr* in section 2.1.3. For the moment, in fact, we will consider streams to be ordinary lists, and *cons-stream*, *head*, and *tail* to be simply alternative names for *cons*, *car*, and *cdr*. This view of streams will be adequate until section 3.4.3, when we will concern ourselves with the efficiency of using streams to represent large aggregates of data.

Computing with streams

Now we can reformulate the two procedures above to match the signal flow diagrams. For *sum-odd-squares*, we need to construct a stream that enumerates the leaves of the tree, to filter a stream for oddness, to square the elements of a stream, and to sum the elements of a stream. We can enumerate the leaves of a tree as follows:

```
(define (enumerate-tree tree)
  (if (leaf-node? tree)
      (cons-stream tree the-empty-stream)
      (append-streams (enumerate-tree (left-branch tree))
                      (enumerate-tree (right-branch tree)))))
```

Append-streams here is a procedure that takes two streams as arguments and produces a

stream that contains all the elements of its first argument followed by all the elements of its second argument, as follows:[21]

```
(define (append-streams s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (head s1)
                   (append-streams (tail s1) s2))))
```

To filter a stream for oddness, we proceed as follows:

```
(define (filter-odd s)
  (cond ((empty-stream? s) the-empty-stream)
        ((odd? (head s))
         (cons-stream (head s) (filter-odd (tail s))))
        (else (filter-odd (tail s)))))
```

To square every element of a stream we can use

```
(define (map-square s)
  (if (empty-stream? s)
      the-empty-stream
      (cons-stream (square (head s))
                   (map-square (tail s)))))
```

And we can sum the elements of a stream with the following procedure:

```
(define (accumulate-+ s)
  (if (empty-stream? s)
      0
      (+ (head s) (accumulate-+ (tail s)))))
```

Now that we have these pieces, we can use them to reorganize the *sum-odd-squares* computation to correspond to the signal flow diagram.

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

With a few more building blocks, we can reformulate the *odd-fibs* procedure in the same way. We need to enumerate an interval of the integers to form a stream. We do this using the following procedure, which returns a stream of consecutive integers from *low* through *high*:

```
(define (enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (enumerate-interval (1+ low) high))))
```

The following procedure applies *fib* to each element of a stream to obtain the stream of corresponding Fibonacci numbers:

---

[21] Notice that if we consider streams to be ordinary lists, writing *cons* for *cons-streams*, *car* for *head*, and so on, then *append-streams* is precisely the *append* procedure that we saw in section 2.2.1 of Chapter 2 and in exercise 3-11 of this chapter.

```
(define (map-fib s)
  (if (empty-stream? s)
      the-empty-stream
      (cons-stream (fib (head s))
                   (map-fib (tail s)))))
```

The next procedure accumulates the items in a stream to form a list, by successively applying *cons*.

```
(define (accumulate-cons s)
  (if (empty-stream? s)
      '()
      (cons (head s) (accumulate-cons (tail s)))))
y
```

Now we can rewrite *odd-fibs* as follows:

```
(define (odd-fibs n)
  (accumulate-cons
    (filter-odd
      (map-fib
        (enumerate-interval 1 n)))))
```

This may seem to be a lot of work merely to write two simple procedures. But now that the two programs have similar structures, we can mix and match the various pieces of our programs to construct other programs. For example, we can construct a list of the squares of the first *n* Fibonacci numbers as follows:

```
(define (list-square-fibs n)
  (accumulate-cons
    (map-square
      (map-fib
        (enumerate-interval 1 n)))))
```

3.4.2. Higher Order Procedures for Streams

We have seen how to capture the commonality in two simple procedures by rewriting them in terms of stream operations. But the two procedures have even more in common than we have yet shown. For example, the two accumulation procedures *accumulate-+* and *accumulate-cons* differ only in the method used to accumulate the results and the initial value for beginning accumulation. Thus we can express both of these procedure in terms of a general *accumulate* abstraction. In Chapter 1, section 1.3, we saw how to formulate such abstractions as higher order procedures. Applying this technique, we can write a general *accumulate* procedure that takes as arguments a method used to combine items, an initial value, and a stream to be accumulated:

```
(define (accumulate combiner initial-value stream)
  (if (empty-stream? stream)
      initial-value
      (combiner (head stream)
                (accumulate combiner
                            initial-value
                            (tail stream)))))
```

Many operations can be expressed in terms of accumulations. For example, we can add the elements in a stream with

```
(define (sum-stream stream)
  (accumulate + 0 stream))
```

or we can multiply the elements in a stream with

```
(define (product-stream stream)
  (accumulate * 1 stream))
```

Our *accumulate-cons* operation can be accomplished by

```
(define (cons-up-stream stream)
  (accumulate cons '() stream))
```

Evaluating a polynomial

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

at a given value of x can also be formulated as an accumulation. We evaluate the polynomial using a well-known algorithm called *Horner's rule*, which structures the computation as

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

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.[22] If we assume that the coefficients of the polynomial are arranged in a stream, from a_0 through a_n, then we can express Horner's rule as an accumulation along the coefficient stream:

```
(define (horner-eval x coefficient-stream)
  (define (add-term coeff higher-terms)
    (+ coeff (* x higher-terms)))
  (accumulate add-term
              0
              coefficient-stream))
```

The idea of this procedure is that, for each coefficient in the stream, we multiply the (already accumulated) higher terms by x, and add in the new coefficient.

Maps and filters

The examples above show how a single abstraction, *accumulate*, can capture many different operations on streams. We can define other abstractions in a similar manner. The *map* procedure generalizes the *map-square* and *map-fib* procedures used above in section 3.4.1. *Map* takes a procedure and a stream as arguments, and generates the stream formed

---

[22] According to Knuth [24], Horner's 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. Notice that Horner's rule evaluates the polynomial using fewer additions and multiplications than does the straightforward method of first computing $a_n x^n$, then adding $a_{n-1} x^{n-1}$, and so on. In fact, it is possible to prove that *any* algorithm for evaluating arbitrary polynomials must use at least as many additions and multiplications as does Horner's rule, and thus Horner's rule is an *optimal algorithm* for polynomial evaluation. This fact 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 [4] provides an overview of these and other results about optimal algorithms.

by applying the procedure to each item in the input stream:

```
(define (map proc stream)
  (if (empty-stream? stream)
      the-empty-stream
      (cons-stream (proc (head stream))
                   (map proc (tail stream)))))
```

The other general operation we used was to filter a stream, extracting those elements that satisfy a given predicate:

```
(define (filter pred stream)
  (cond ((empty-stream? stream) the-empty-stream)
        ((pred (head stream))
         (cons-stream (head stream)
                      (filter pred (tail stream))))
        (else (filter pred (tail stream)))))
```

By combining filters, maps, and accumulators, we can express new operations such as

```
(define (product-of-squares-of-odd-elements stream)
  (accumulate *
              1
              (map square
                   (filter odd? stream))))
```

We can also formulate conventional "data processing" applications in terms of streams. For example, suppose we have a stream of personnel records, and we want to find the salary of the highest-paid programmer. Assume that we have selectors *job* and *salary* that return the required information from a record. Then we can write

```
(define (salary-of-highest-paid-programmer record-stream)
  (define (programmer? record)
    (eq? (job record) 'programmer))
  (accumulate max
              0
              (map salary
                   (filter programmer?
                           record-stream))))
```

These two examples give just a hint of the vast range of operations that can be expressed in this way.[23]

Another useful abstraction similar to *map* is *for-each*, which applies a procedure to every item in a stream, but does not accumulate the results to form an output stream:

---

[23] As part of a 1978 Ph.D. thesis on program analysis, Richard Waters [50] developed a program that automatically analyzes traditional FORTRAN programs, viewing them in terms of maps, filters, and accumulations. He found that fully 60 percent of the code in the FORTRAN Scientific Subroutine Package fits neatly into this paradigm. Indeed, one of the reasons for the success of Lisp as a programming language is that lists are an excellent standard medium for expressing ordered collections so that they can be manipulated in this way. The programming language APL obtains much of its power and appeal by a similar choice. In the case of APL all data is represented as square arrays. There is a universal and convenient set of generic operators for all sorts of array operations.

```
(define (for-each proc stream)
  (if (empty-stream? stream)
      'done
      (sequence (proc (head stream))
                (for-each proc (tail stream)))))
```

For example, to print a stream, we can use[24]

```
(define (print-stream s)
  (for-each print s))
```

Nested accumulations

Consider the following problem: Given a positive integer *n*, find all pairs of distinct positive integers *i* and *j* less than or equal to *n* such that *i+j* is prime. For example, if *n* is 6, then the pairs are

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

Assume we use *(list i j)* to represent the pair (*i*, *j*) and that we use a predicate *prime?* that tests primality.

One way to organize this computation is as a nested accumulation over streams. For each *i* in the stream *(enumerate-interval 1 n)*, we map along the stream of integers 1 from through *i-1* as follows: For each *j* in the stream *(enumerate-interval 1 (-1+ i))*, we test whether *i+j* is prime. If so, we generate the stream containing the single item *(list i j)*; if not, we generate the empty stream. Appending together all of these (empty or 1 element) streams produces a stream of pairs for each *i*. Appending together all of *these* streams produces the required stream of all pairs *i*, *j*

We can express this nested accumulation in terms of a procedure called *flatmap*, which takes as arguments an input stream and a procedure to be applied to each item of the input stream. The procedure is assumed to return a *stream* each time it is applied to an argument. *Flatmap* applies the procedure to each element of the input stream and combines all the elements of the resulting streams to form a single stream.

Using *flatmap*, we can generate all pairs of integers *i* and *j* less then *n* such that *i+j* is prime, as follows:

```
(define (prime-sum-pairs n)
  (flatmap (lambda (i)
             (flatmap (lambda (j)
                        (if (prime? (+ i j))
                            (singleton (list i j))
                            the-empty-stream))
                      (enumerate-interval 1 (-1+ i))))
           (enumerate-interval 1 n)))
```

Notice that the *lambda (j)* produces a stream for each *j*, and that these streams are combined to form a single stream by the inner *flatmap*. This is the stream returned for each

---

[24] If streams are represented as lists, the interpreter will automatically print them in standard list notation. If we use other representations for streams, as we will in section 3.4.3, then a procedure such as *print-stream* becomes necessary, unless we build some conventional way for printing streams into the *print* primitive.

i by the *(lambda (i))*, and the streams for all the *i*s are combined by the outer *flatmap*.

The *singleton* procedure generates a stream that contains a single designated item:

```
(define (singleton s)
  (cons-stream s the-empty-stream))
```

Flatmap can be defined as follows:[25]

```
(define (flatmap f s)
  (if (empty-stream? s)
      the-empty-stream
      (let ((s1 (f (head s))))
        (if (empty-stream? s1)
            (flatmap f (tail s))
            (cons-stream (head s1)
                         (append-streams (flatmap f (tail s))
                                         (tail s1)))))))
```

Another nested accumulation that we can handle in the same way is to find all triples of distinct positive integers *i*, *j*, *k*, less than or equal to a given integer *n*, which sum to a given integer *s*. For example, with *n* equal to 9 and *s* equal to 15 the triples are:[26]

```
(6 5 4)  (7 5 3)  (7 6 2)  (8 4 3)
(8 5 2)  (8 6 1)  (9 4 2)  (9 5 1)
```

We can organize this computation using *flatmap*, just as in *prime-sum-pairs*. We map along the interval from 1 to *n*, generating a stream for each *i* in the interval. The stream for each *i* is obtained by mapping along the interval from 1 to *i-1*, generating a stream for each *j* in the interval. The stream for each *j* is obtained by mapping along the interval from 1 to *j-1*, checking the condition

```
(lambda (k) (= (+ i j k) s))
```

and generating either the one-element stream containing the triple *(list i j k)*, or else the empty-stream. Here is the complete procedure:

---

[25] In section 3.4.4, we will learn how to work with infinitely long streams. We can use the methods of the present section to perform nested accumulations over infinitely long streams, provided that we make a small change to the *flatmap* procedure. See exercise 3-44. This approach to accumulation was shown to us by David Turner, whose language KRC provides an elegant formalism for dealing with accumulations. The examples in this section (see also exercise 3-37) are adapted from Turner's paper [49].

[26] These triples represent all the wins in tic-tac-toe if we number the 9 positions on a tic-tac-toe board as in the following "magic square":

```
 4 | 9 | 2
---|---|----
 3 | 5 | 7
---|---|---
 8 | 1 | 6
```

```
(define (triples n s)
  (flatmap
   (lambda (i)
     (flatmap (lambda (j)
                (flatmap (lambda (k)
                           (if (= (+ i j k) s)
                               (singleton (list i j k))
                               the-empty-stream))
                         (enumerate-interval 1 (-1+ j))))
              (enumerate-interval 1 (-1+ i))))
   (enumerate-interval 1 n)))
```

We can make these nested accumulations easier to write by providing some syntactic sugar. One possibility is to use a special form called `collect`, which is defined so that

```
(collect <result>
         ((<v1> <set1>)
          (<v2> <set2>)
          ...
          (<vn> <setn>))
         <restriction>)
```

is equivalent to

```
(flatmap (lambda (<v1>)
           (flatmap (lambda (<v2>)
                      ...
                      (flatmap (lambda (<vn>)
                                 (if <restriction>
                                     (singleton <result>)
                                     the-empty-stream))
                               <setn>))
                    ...
                    <set2>))
         <set1>)
```

Using `collect`, we can rewrite `prime-sum-pairs` as

```
(define (prime-sum-pairs n)
  (collect (list i j)
           ((i (enumerate-interval 1 n))
            (j (enumerate-interval 1 (-1+ i))))
           (prime? (+ i j))))
```

and we can rewrite `triples` as

```
(define (triples n s)
  (collect (list i j k)
           ((i (enumerate-interval 1 n))
            (j (enumerate-interval 1 (-1+ i)))
            (k (enumerate-interval 1 (-1+ j))))
           (= (+ i j k) s)))
```

These nested accumulations are similar to the "nested loops over index variables" found in many programming languages. We can interpret the meaning of the `collect` as

for each ⟨v_1⟩ in ⟨set_1⟩ and
  for each ⟨v_2⟩ in ⟨set_2⟩ and
    ...
      for each ⟨v_n⟩ in ⟨set_n⟩
        if all the ⟨v_i⟩ satisfy the ⟨*restriction*⟩
          then accumulate the ⟨*result*⟩

We can also perform accumulations over streams other than those that enumerate intervals. For instance, consider the problem of generating all the permutations of a set S of items; that is, all the ways of ordering the items in the set. For instance, the permutations of *(a b c)* are:

```
(a b c) (a c b) (b a c) (b c a) (c a b) (c b a)
```

Here is a plan for generating the permutations of S. For each each item x in S, we recursively generate the stream of all permutations p of $S-x$. (The set $S-x$ is the set of all elements of S, excluding x.) Then for each such permutation p, we adjoin x to the front of p. This yields, for each x in S, the stream of permutations of S that begin with x, and combining these streams gives all the permutations of S. Thus we can reduce the problem of generating permutations of sets of n items to the problem of generating the permutations of sets of $n-1$ items. If S itself is represented as a stream of elements, then we can reduce the problem of generating the permutations of S to the problem of generating the permutations on shorter and shorter streams. This leads to the following procedure:[27]

```
(define (permutations S)
  (if (empty-stream? S)
      (singleton the-empty-stream)
      (flatmap (lambda (x)
                 (flatmap (lambda (p)
                            (singleton (cons-stream x p)))
                          (permutations (remove x S))))
               S)))
```

Alternatively, we can rewrite the *permutations* procedure using *collect*[28]

```
(define (permutations S)
  (if (empty-stream? S)
      (singleton the-empty-stream)
      (collect (cons-stream x p)
               ((x S)
                (p (permutations (remove x S)))))))
```

The *remove* procedure used in *permutations* returns all of the items in a given stream, except for a given item. This can be expressed as a simple filter:

---

[27] In the terminal case, when we have worked our way down to *the-empty-stream*, which represents a set of no elements, we use *(singleton the-empty-stream)* to generate as "the permutations" a stream with one item, namely the set with no elements.

[28] Observe that this *collect* form has no ⟨*restriction*⟩ clause. We assume that *collect* is defined so as to not impose a restriction if none is specified.

```
(define (remove item stream)
  (filter (lambda (x) (not (equal? x item)))
          stream))
```

Exercise 3-34: Consider the following alternative version of the *accumulate* procedure:

```
(define (left-accumulate combiner initial-value stream)
  (if (empty-stream? stream)
      initial-value
      (left-accumulate combiner
                       (combiner initial-value (head stream))
                       (tail stream))))
```

Does *(left-accumulate + 0 x)* return the same result as *(accumulate + 0 x)* for any stream of numbers *x*? Does *(left-accumulate cons '() x)* return the same result as *(accumulate cons '() x)* for any stream *x*? In general, for which *combiner* procedures will *accumulate* and *left-accumulate* give the same result?

Exercise 3-35: The procedure *accumulate-n* is similar to *accumulate*, except that it takes as its third argument a stream of streams, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the streams, all the second elements of the streams, and so on, and returns a stream of the results. For instance, if *S* is a stream containing four streams:

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

Then *(accumulate-n + 0 S)* should return the stream *(22 26 30)*. Fill in the missing expressions $\langle exp_1 \rangle$ and $\langle exp_2 \rangle$ in the following definition of *accumulate-n*:

```
(define (accumulate-n op init streams)
  (if (empty-stream? (head streams))
      the-empty-stream
      (cons-stream
       (accumulate op init <exp1>)
       (accumulate-n op init <exp2>))))
```

Exercise 3-36: Suppose we represent vectors $v = (v_i)$ as streams of numbers, and matrices $m = (m_{ij})$ as streams of vectors (the rows of the matrix). For example, the matrix

```
1  2  3  4
4  5  6  6
6  7  8  9
```

is represented as the stream *((1 2 3 4) (4 5 6 6) (6 7 8 9))*. With this representation, we can use stream operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are

- *(dot-product v w)* is the sum $\Sigma_i v_i w_i$
- *(matrix-times-vector m v)* is the vector *t*, where $t_i = \Sigma_j m_{ij} v_j$
- *(matrix-times-matrix m n)* is the matrix *p*, where $p_{ij} = \Sigma_k m_{ik} n_{kj}$
- *(transpose m)* is the matrix *n*, where $n_{ij} = m_{ji}$

Fill in the missing expressions in the following procedures for computing these operations. (The procedure *accumulate-n* is defined in exercise 3-35.)

```
(define (dot-product v w)
  (accumulate + 0 <???>))

(define (matrix-times-vector m v)
  (map <???> m))

(define (transpose mat)
  (accumulate-n <???> <???> mat))
```

```
(define (matrix-times-matrix m n)
  (let ((cols (transpose n)))
    (map <???> m)))
```

Exercise 3-37: A famous puzzle, called the *eight queens problem*, asks how to place 8 queens on a chess board so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). Here is one possible solution:

```
- - - - - x - -
- - x - - - - -
x - - - - - - -
- - - - - - x -
- - - - x - - -
- - - - - - - x
- x - - - - - -
- - - x - - - -
```

One way to solve the problem is to work across the chess board, placing a queen in each column. Assuming we have already placed $k-1$ queens, then we must place the k-th queen in a position where it does not check any of the queens already on the board. We can formulate this approach using a recursive plan: Assume we have already generated the stream of *all* possible ways to place $k-1$ queens in the first $k-1$ columns of the board. We extend each of these ways by generating all the rows q for which it is safe to place a queen in the q-th row and the k-th column. This produces the stream of all ways to place k queens in the first k columns. We implement this solution as a procedure *queens*, which returns a stream of all solutions to the problem of placing n queens on an $n \times n$ chess board. This calls a procedure *queen-cols*, which returns the stream of all ways to place queens in first k columns of a board of specified size.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (singleton the-empty-stream)
        (collect (cons q b)
                 ((b (queen-cols (-1+ k)))
                  (q (enumerate-interval 1 board-size)))
                 (safe? q b))))
  (queen-cols board-size))
```

In the *collect* used by *queen-cols*, b is a way to place $k-1$ queens in the first $k-1$ columns, and q is a proposed row in the kth column in which to place a new queen. You must complete the program by implementing the procedure *safe?* Also, rewrite the *queen-cols* procedure without using *collect*.

Exercise 3-38: Louis Reasoner is having a terrible time doing exercise 3-37. His *queens* procedure seems to work, but it runs extremely slowly. (Louis never does manage to wait long enough for it to solve even the 6×6 case.) When he asks Eva Lu Ator for help, she points out that he has interchanged two lines in the *queen-cols* procedure, writing the *collect* form as

```
(collect (cons q b)
         ((q (enumerate-interval 1 board-size))
          (b (queen-cols (-1+ k))))
         (safe? q b))))
```

Explain why this interchange makes the program run slowly. Estimate how long it will take Louis's program to solve the eight queens problem, assuming that the program in exercise 3-37 solves the problem in time T.

3.4.3. Streams and Delayed Evaluation

As we have seen, streams can serve as a standard interface between program modules. By using streams, we can formulate powerful abstractions that capture a wide variety of program operations in a manner that is both succinct and elegant. Unfortunately, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations -- or at least this will be the case if we represent streams as ordinary lists, with *cons-stream*, *head*, and *tail* defined, respectively, as *cons*, *car*, and *cdr*.

If we represent streams as lists, our programs must construct and copy possibly huge data structures at every step of a stream 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:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (1+ count) (+ count accum)))
          (else (iter (1+ count) accum))))
  (iter a 0))
```

The second program performs the same computation using streams:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, with streams represented as lists, the *filter* in the second program cannot do any testing until *enumerate-interval* has constructed a complete list of the numbers in the interval. The *filter* itself generates another large 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.

We can see an even more extreme example of inefficiency if we imagine computing the second prime in the interval from 10,000 to 1,000,000 interval by evaluating the expression

```
(head (tail (filter prime?
                    (enumerate-interval 10000 1000000))))
```

In principle, this expression does find the second prime, but the computational overhead seems 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.

We will now see how, by changing the representation of streams, we can achieve the best of both worlds: We can use the elegant stream formulation, while preserving the efficiency of incremental computation. The basic idea is that we will arrange for *cons-stream* 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 enable the consumer to access 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 streams, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

To make streams behave in this way, we will arrange for the tail of a stream to be evaluated, not when the stream is constructed by *cons-stream*, but rather when the tail is accessed by the *tail* procedure. This implementation choice is reminiscent of our discussion of rational numbers in Chapter 2, where we saw in section 2.1.2 that we can implement rational numbers so that the reduction of numerator and denominator to lowest terms is performed either at construction time or 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 *car* and the *cdr* are evaluated at construction time. With streams, the *tail* is evaluated at selection time.

Our implementation of streams will be based on a special form called *delay*. Evaluating the form *(delay <exp>)* does not evaluate the expression *<exp>*, but rather returns a so-called *delayed object*, which we can think of as a "promise" to evaluate *<exp>* at some future time. As a companion to *delay*, we have an operator called *force*, which takes a delayed object as argument, and performs the evaluation -- in effect, forcing the *delay* to fulfill its promise . We will see below how *delay* and *force* can be implemented, but first let us use these to construct streams.

Cons-stream is a special form defined so that

```
(cons-stream <a> <b>)
```

is equivalent to

```
(cons <a> (delay <b>))
```

What this means is that we will construct streams using pairs, but rather than placing the value of the *tail* into the *cdr* of the pair, we will put there a promise to compute the *tail* if it is ever requested. *Head* and *tail* can now be defined as procedures:

```
(define (head stream) (car stream))

(define (tail stream) (force (cdr stream)))
```

In other words, *head* selects the *car* of the pair that was constructed by *cons-stream*, while *tail* selects the *cdr* of the pair and evaluates the delayed expression found there to obtain the tail.[29]

---

[29] Observe that although *head* and *tail* can be defined as procedures, *cons-stream* must be a special form. If *cons-stream* were a procedure, then, according to our model of evaluation, evaluating *(cons-stream <a> <b>)* would automatically cause *<b>* to be evaluated, which is precisely what we do not want to happen. For the same reason, *delay* must be a special form. See exercise 3-40.

The stream implementation in action

To see how this implementation behaves, let us analyze the "outrageous" prime computation

```
(head (tail (filter prime?
                    (enumerate-interval 10000 1000000))))
```

and see that, in fact, it works efficiently. The evaluation begins by calling *enumerate-interval* with the arguments 10,000 and 1,000,000. Recall that *enumerate-interval* is defined as

```
(define (enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (enumerate-interval (1+ low) high))))
```

so the result, formed by the *cons-stream*, is

```
(cons 10000 (delay (enumerate-interval 10001 1000000)))
```

that is, the result is a stream represented as a pair whose *car* is 10000 and whose *cdr* is a promise to enumerate more of the interval if so requested. This stream is now filtered for primes, using the *filter* procedure:

```
(define (filter pred stream)
  (cond ((empty-stream? stream) the-empty-stream)
        ((pred (head stream))
         (cons-stream (head stream)
                      (filter pred (tail stream))))
        (else (filter pred (tail stream)))))
```

Filter tests the *head* of the stream (the *car* of the pair, which is 10,000) and finds that this is not prime. So *filter* examines the *tail* of its input stream. The call to *tail* forces evaluation of the delayed *enumerate-interval*, which now returns

```
(cons 10001 (delay (enumerate-interval 10002 1000000)))
```

Filter now looks at the *head* of this stream, 10,001, sees this is not prime either, forces another tail, and so on, until *enumerate-interval* yields the prime 10,007, whereupon *filter*, according to its definition, returns

```
(cons-stream (head stream)
             (filter pred (tail stream)))
```

which in this case is

```
(cons 10007
      (delay
       (filter prime?
               (cons 10008
                     (delay (enumerate-interval 10009
                                                1000000))))))
```

This result is now passed to *tail* in our original expression. *Tail* forces the delayed *filter*, which keeps forcing the delayed *enumerate-interval* until it finds the next prime, which is 10,009. So, finally, the result passed to *head* in our original expression is

```
(cons 10009
      (delay
       (filter prime?
               (cons 10010
                     (delay (enumerate-interval 10011
                                                1000000))))))
```

Head returns 10,009, and the computation is complete. Notice that 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 this 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 procedures. We write procedures as if the streams existed "all at once" when, in reality, the computation is performed incrementally, as in traditional programming styles.

Implementing DELAY and FORCE

Although *delay* and *force* may seem like powerful and complex operations, their implementation is really quite straightforward. What we need for *delay* is to package an expression so that it can be evaluated later on demand, and we can accomplish this simply by treating the expression as the body of a procedure. Which is to say, we can regard *delay* as a special form such that *(delay <exp>)* is syntactic sugar for

```
(lambda () <exp>)
```

Force simply calls the procedure (of no arguments) produced by *delay*, so we can implement *force* as a procedure:

```
(define (force delayed-object)
  (delayed-object))
```

This implementation suffices for *delay* and *force* to work as advertised, but there is an important optimization that we can make. 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 3-42.) The solution is to build a delayed object so that, the first time it is forced, it stores the value that is computed. Subsequent forcings will simply return the stored value without repeating the computation. In other words, we implement *delay* as a special purpose *memoized* procedure, similar to the one described in exercise 3-24. One way to accomplish this is to use the following procedure, which takes as argument a procedure (of no arguments) and returns a memoized version of the procedure:

```
(define (memo-proc proc)
  (let ((already-computed? nil) (result nil))
    (lambda ()
      (if (not already-computed?)
          (sequence (set! result (proc))
                    (set! already-computed? (not nil))
                    result)
          result))))
```

Delay is then defined so that *(delay <exp>)* is equivalent to

```
(memo-proc (lambda () <exp>))
```

and *force* is as previously defined.[30]

Exercise 3-39: In order to take a closer look at delayed evaluation, we will use the following procedure, which simply returns its argument after printing it:

```
(define (show x)
  (print x)
  x)
```

The next procedure, similar to the *nth* procedure of section 2.2.1, extracts a given item from a stream:

```
(define (nth-stream n s)
  (if (= n 0)
      (head s)
      (nth-stream (-1+ n) (tail s))))
```

What does the interpreter print in response to evaluating each expression in the following sequence?[31]

```
(define x (map show (enumerate-interval 0 10)))

==>(nth-stream 5 x)
<printed response>

==>(nth-stream 7 x)
<printed response>
```

Exercise 3-40: Ben Bitdiddle has become severely annoyed while doing exercise 3-36, because he has realized that he can not use *cons-stream* as an argument to a higher-order procedure. (Supplementary exercise: Why does Ben want to do this in exercise 3-36?) To make the best of a bad situation, he has decided to use *cons* instead. To explore the effect that this will have on his programs, he uses the *show* procedure of exercise 3-39 to compare two procedures for copying streams. The first accumulates with *cons-stream*, but since Ben can't use *accumulate* explicitly, he writes out the accumulation pattern

```
(define (copy-stream s)
  (if (empty-stream? s)
      the-empty-stream
      (cons-stream (head s) (copy-stream (tail s)))))
```

The second program is the accumulation that Ben would have liked to have written, except that he writes *cons* in place of *cons-stream*:

```
(define (*copy-stream s)
  (accumulate cons the-empty-stream s))
```

---

[30] 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 [28] in 1965. Delayed evaluation for streams was introduced into Lisp by Freidman and Wise [11] in 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 thunks* and to the optimized versions as *call-by-need thunks*.

[31] Exercises such as these are valuable for testing our understanding of how *delay* works. On the other hand, intermixing delayed evaluation with printing and, even worse, with assignment, is extremely confusing, and professors of courses on computer languages have traditionally tormented their students with examination questions such 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.

What does Ben see printed in response to each of the following expressions?

```
==>(sequence (copy-stream (map show (enumerate-interval 1 10)))
             'done)
<printed response>

==>(sequence (copy-stream (map show (enumerate-interval 1 10)))
             'done)
<printed response>
```

3.4.4. Infinitely Long Streams

We have now seen how to support the illusion of manipulating streams as "completed entities" even though, in actuality, we compute only as much of the stream as we need to access. We can exploit this technique to efficiently represent sequences as streams, even if the sequences are very long. Better yet, we can use streams to represent sequences that are *infinitely* long. For instance, consider the following definition of the stream of positive integers:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (1+ n))))

(define integers (integers-starting-from 1))
```

This makes sense because *integers* will be a pair whose *car* is 1 and whose *cdr* 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, so 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:

```
(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (filter (lambda (x) (not (divisible? x 7)))
          integers))
```

and we can use this to find the 100th integer not divisible by 7:[32]

```
==>(nth-stream 100 no-sevens)
117
```

In analogy with *integers*, we can define the infinite stream of Fibonacci numbers:

```
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

Fibs is a pair whose *car* is 0 and whose *cdr* is a promise to evaluate *(fibgen 1 1)*, which, when we evaluate it, will produce a pair whose *car* is 1 and whose *cdr* is a promise to evaluate *(fibgen 1 2)*, and so on.

---

[32] The *nth-stream* procedure was defined in exercise 3-39.

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*.[33] 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*, we form a stream whose *head* is the *head* of *S* and whose *tail* is obtained by filtering all multiples of the *head* of *S* out of the *tail* of *S* and sieving the result. This process is readily described in terms of stream operations:

```
(define (sieve stream)
  (cons-stream
   (head stream)
   (sieve (filter
           (lambda (x) (not (divisible? x (head stream))))
           (tail stream)))))

(define primes (sieve (integers-starting-from 2)))
```

Now to find a particular prime, we need only ask for it:

```
==>(nth-stream 50 primes)
233
```

It is interesting to contemplate the signal processing system set up by *sieve*, shown in the "Henderson diagram" in figure 3-28.[34] The input stream feeds into an "un-*cons*-er" that separates the *head* of the stream from the *tail*. The *head* is used to construct a divisibility filter, through which the *tail* is passed, and the output of the filter is fed to another sieve box. Then the original *head* is *cons*-ed onto the output of the internal sieve to give the output stream. So not only is the stream infinite, but the signal processor is also infinite, because the sieve contains a sieve within it.

---

[33] Eratosthenes was a third century B.C. Alexandrian Greek philosopher, who is famous for giving the first accurate estimate of the circumference of the earth, which he computed by observing shadows cast at noon on the day of the summer solstice. Eratosthenes' sieve method, although ancient, has formed the basis for special-purpose hardware "sieves" that, until very recently, were the most powerful tools in existence for locating large primes. Over the past few years, these methods have been superseded by outgrowths of the probabilistic techniques discussed in section 1.2.6 of Chapter 1.

[34] 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.

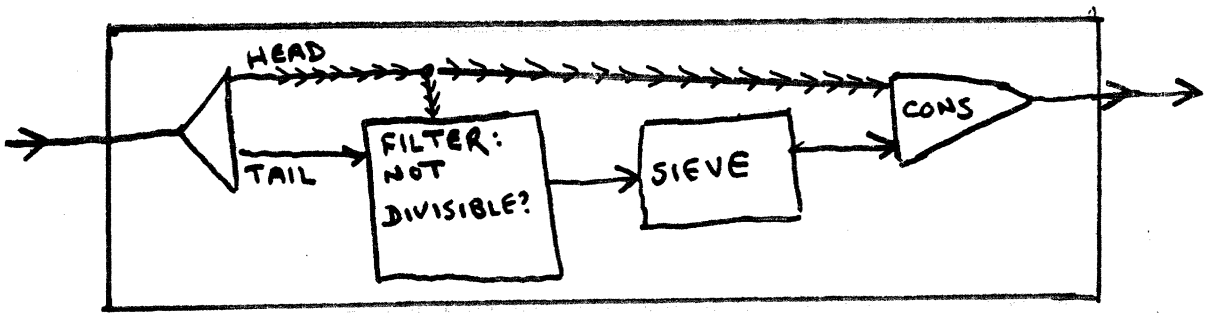


Figure 3-28: The prime sieve viewed as a signal processing system.

The *integers* and *fibs* streams above were defined by specifying "generating" procedures 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:

```
(define ones (cons-stream 1 ones))
```

This works much like the definition of a recursive procedure: *ones* is a pair whose *car* is 1 and whose *cdr* is a promise to evaluate *ones*. Evaluating the *cdr* gives us again a 1 and a promise to evaluate *ones*, and so on.

We can do more interesting things by using procedures such as *add-streams*, which produces the element-wise sum of two given streams:

```
(define (add-streams s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else (cons-stream (+ (head s1) (head s2))
                           (add-streams (tail s1) (tail s2))))))
```

Now we can define the integers as:

```
(define integers (cons-stream 1 (add-streams ones integers)))
```

This works by defining *integers* to be a stream whose *head* is 1 and whose *tail* is the sum of *integers* and *ones*. So the *head* of the *tail* is the *head* of *integers* plus 1, or 2. The third element of *integers* is 1 plus the second element of *integers*, or 3. And so on. Notice that 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.

We can define the Fibonacci numbers in the same style:

```
(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (tail fibs) fibs))))
```

This definition says that *fibs* is a stream beginning with 0 and 1, such that the stream can be generated by adding it to itself shifted by one place:

```
0  1  1  2  3  5  8 13 21 ...
   0  1  1  2  3  5  8 13 ...
---------------------------
   1  2  3  5  8 13 21 34 ...
```

Another useful procedure in formulating such stream definitions is *scale-stream*, which multiplies each item in a stream by a given constant:

```
(define (scale-stream c stream)
  (map (lambda (x) (* x c)) stream))
```

For example,

```
(define double (cons-stream 1 (scale-stream 2 double)))
```

produces the stream of powers of 2

```
1, 2, 4, 8, 16, 32, ......
```

We can also give an alternative definition of the stream of primes. The idea is that we start with the integers and filter them by testing for primality. We will need the first prime, 2, to get started:

```
(define primes
  (cons-stream 2 (filter prime? (tail (tail integers)))
```

This looks easy to understand, except that we will test whether a number *n* is prime by checking if *n* is divisible by a prime less or equal to| than the square root of *n*:

```
(define (prime? n)
  (define (iter ps)
    (cond ((> (square (head ps)) n) 't)
          ((divisible? n (head ps)) the-empty-stream)
          (else (iter (tail ps)))))
  (iter primes))
```

This is a recursive definition, since *primes* is defined in terms of the *prime?* predicate, which itself uses the *primes* stream. The reason this procedure works is that, at any point, enough of the *prime* stream has been generated to test the primality of the numbers we need to check next.[35]

Exercise 3-41: 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 increasingly fewer of them fit the requirement. Instead, let us call the required stream of numbers *S*, and notice the following facts about it.

- *S* begins with a 1.
- The elements of *(scale-stream 2 S)* are also elements of *S*.
- Similarly for *(scale-stream 3 S)* and *(scale-stream 5 S)*.

---

[35] This is a very subtle point, and relies on the fact that $p_{n+1} \leq p_n^2$, where p_k denotes the *k*-th prime. Estimates such as these are very difficult to establish. The ancient proof by Euclid that there are an infinite number of primes shows that $p_{n+1} \leq p_1 p_2 \cdots p_{n-1} p_n + 1$, 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 the book by Hardy and Wright [17] section 22.3.

- These are all the elements of *S*.

So all we have to do is to combine elements from these four sources. For this we define a procedure, *merge*, which combines two ordered streams into one ordered result stream, eliminating repetitions.

```
(define (merge s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else
         (let ((h1 (head s1))
               (h2 (head s2)))
           (cond ((< h1 h2) (cons-stream h1 (merge (tail s1) s2)))
                 ((> h1 h2) (cons-stream h2 (merge s1 (tail s2))))
                 (else (cons-stream h1 (merge (tail s1) (tail s2)))))))))
```

Then the required stream may be constructed using *merge*, as follows:

```
(define S (cons-stream 1 (merge <???> <???>))
```

Fill in the missing expressions in the places marked *<???>* above.

Exercise 3-42: How many additions are performed when we compute the *n*-th Fibonacci number, using the definition of *fibs* based on the *fibgen* procedure? Show that the number of additions would be exponentially greater if we implemented *(delay <exp>)* simply as *(lambda () <exp>)*, without using the optimization provided by the *memo-proc* procedure as described in section 3.4.3.[36]

Exercise 3-43: Give an interpretation of the stream computed by the following procedure:

```
(define (expand num den radix)
  (cons-stream (quotient (* num radix) den)
               (expand (remainder (* num radix) den) den radix)))
```

What are the successive elements produced by *(expand 1 7 10)*? How about *(expand 3 8 10)*?

Exercise 3-44: In section 3.4.2, we learned about programs that perform nested accumulations over streams. For example, the following program generates all pairs of elements *i*, *j*, where *i* runs through a stream *S1* and *j* runs through a stream *S2*:

```
(define (pairs S1 S2)
  (collect (list i j)
           ((i S1)
            (j S2))))
```

As we saw, the *collect* form used here syntactic sugar for

```
(define (pairs S1 S2)
  (flatmap (lambda (x)
             (flatmap (lambda (y)
                        (singleton (list x y)))
                      S2))
           S1))
```

Unfortunately, this method of accumulation is not satisfactory for very long streams (much less for infinite streams). The problem lies in the definition of the *flatmap* procedure:

---

[36] This exercise shows how call-by-need is closely related to ordinary memoization as described in exercise 3-24. 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 tails of the stream.

```
(define (flatmap f s)
  (if (empty-stream? s)
      the-empty-stream
      (let ((s1 (f (head s))))
        (if (empty-stream? s1)
            (flatmap f (tail s))
            (cons-stream (head s1)
                         (append-streams (flatmap f (tail s))
                                         (tail s1)))))))
```

Since *flatmap* appends the streams formed for successive values of the index variables, it will try to generate the entire stream for the first value of an index variable before proceeding to the second. This is unsuitable for infinite streams. For example, if we try to generate all pairs of positive integers using

```
(pairs integers integers)
```

our stream of results will first try to run through *all* pairs of integers with j equal to 1, and hence will never proceed to any other values of j.

To accumulate over infinite streams, we need to devise an order of accumulation that assures that all elements will eventually be reached if we let our program run long enough.[37] An elegant way to accomplish this (shown to us by David Turner) is to modify the definition of *flatmap* to use the following *interleave* procedure in place of *append-streams*.

```
(define (interleave s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (head s1)
                   (interleave s2
                               (tail s1)))))
```

Collect is defined just as before, using this new definition of *flatmap*.

Modify *flatmap* as described, and generate the stream of all pairs of positive integers i, j. What are the first few pairs produced? In this order of evaluation, j grows much faster than i. How large has j become by the time i reaches 10?

Exercise 3-45: Use nested accumulation to generate the stream of triples of positive integers i, j, k such that $i+j > k$. (Hint: If you do this in the most straightforward way, with i, j, and k, each running through all the positive integers, your program will be extremely inefficient. Why? A better method makes use of the fact that k must lie in the interval between 1 and $i+j$.)

Exercise 3-46: Use nested accumulation to generate the stream of all triples of positive integers i, j, and j such that $i > j$ and $i^2 + j^2 = k^3$. (Hint: To get your program to produce results, you will need to make use of an idea like the one given in the hint for exercise 3-45.)

Exercise 3-47: In section 2.4.3, we saw how to implement a polynomial arithmetic system, representing polynomials as lists of terms. In a similar way, we can work with *power series*, such as

$$\sin x = 1 - x + \frac{x^3}{3\cdot 2} - \frac{x^5}{5\cdot 4\cdot 3\cdot 2} + \ldots$$

represented as streams of infinitely many terms. Using the basic constructors and selectors for *terms* defined in section 2.4.3, design a *term-stream* representation for power series, analogous to the *term-list* representation for polynomials. Using this, implement arithmetic operations on power series. In addition, you should be able to integrate power series termwise, using the operation

---

[37] The precise statement of what we want is as follows: If we are accumulating over n streams, then there should be a function F of n variables such that the element of the result corresponding to element i_1 of the first stream, element i_2 of the second stream, ..., element i_n of the nth stream will appear as element number $F(i_1, i_2, \ldots, i_n)$ of the output stream.

```
(define (integrate-term t)
  (let ((new-order (1+ (order t))))
    (make-term (sign t) new-order (divide-coeff (coeff t) new-order))))

(define (integrate series)
  (map integrate-term series))
```

If you have allowed rational numbers as coefficients, then you should be able to generate the series for *sine* and *cosine* starting from a `unit-term` (which represents 1) as follows:

```
(define cosine-series
  (cons-stream unit-term (integrate (negate-series sine-series))))

(define sine-series
  (integrate cosine-series))
```

3.4.5. Streams as Signals

We began our discussion of streams by describing them as computational analogues 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*, which, for an input stream $x=(x_i)$, an *initial value* C, and a small increment dt, accumulates the sum

$$S_i = C + \sum_{j=0}^{i} x_j dt$$

and returns the stream of values $S=(S_i)$. The required procedure is reminiscent of the "implicit style" definition of the stream of integers in section 3.4.4.

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream dt integrand) int)))
  int)
```

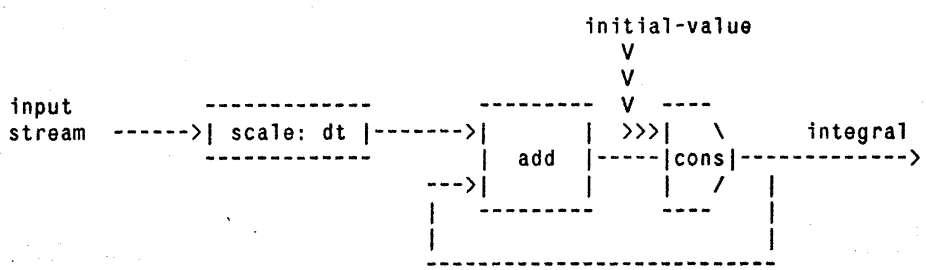
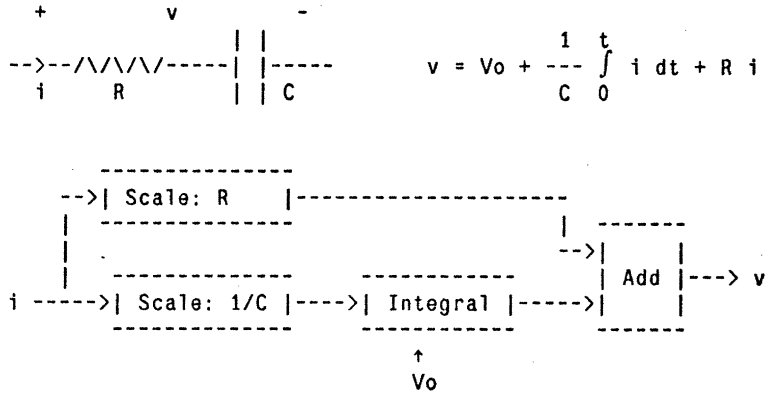


Figure 3-29: The `integral` procedure, viewed as a signal-processing system.

Figure 3-29 is a picture of a signal-processing system that corresponds to the `integral` procedure. The input stream is scaled by `dt` and passed through an adder, whose output is passed back through the same adder. Notice that the `cons-stream` provides the `delay` that permits the internal stream `int` to be defined in terms of itself. (The different style line from

initial-value to the *cons* in the figure indicates that this is a single value transmitted, rather than a stream of values.) 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. In general, *delay* is crucial for modeling signal-processing systems containing loops. Without *delay*, we would be forced to formulate our model so that the inputs to any signal-processing component are fully evaluated before the output can be produced, thus outlawing loops.

Exercise 3-48: We can model electrical systems using streams to represent the values of currents or voltages at consecutive time intervals. For instance, suppose we have an *RC circuit* consisting of a resistor and a capacitor in series, driven by a current source. The voltage response of the system to an injected current *i* is determined by the following formula, whose structure is shown by the accompanying signal-flow diagram:



Write a procedure *RC*, which models this system. *RC* should take as inputs the values of *R*, *C* and *dt*. *RC* should return a procedure that takes as inputs a stream representing the current *i* and an initial value for the capacitor voltage *Vo*, 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

```
(define RC1 (RC 5 1 0.5))
```

to define *RC1* to be a procedure that takes a stream representing the time sequence of currents and an initial capacitor voltage and produces the output stream of voltages.

Exercise 3-49: 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 considered to be 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 to be the corresponding stream of zero-crossings. Alyssa first writes a procedure *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

```
(define (make-zero-crossings input-stream last-value)
  (cons-stream (sign-change-detector (head input-stream) last-value)
               (make-zero-crossings (tail input-stream)
                                    (head input-stream))))

(define 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 a higher-level procedure:

```
(define zero-crossings
  (map-2 sign-change-detector sense-data <expression>))

(define (map-2 proc s1 s2)
  (cons-stream (proc (head s1) (head s2))
               (map-2 proc (tail s1) (tail s2))))
```

Complete the program by supplying the indicated *<expression>*.

Exercise 3-50: Unfortunately, Alyssa's zero-crossing detector in exercise 3-49 proves to be insufficient, because the signal from the sensor is noisy, leading to spurious zero-crossings being produced. Lem E. Tweakit, a hardware specialist, suggests that Alyssa should 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 implements the idea, altering Alyssa's program from exercise 3-49 as follows:

```
(define (make-zero-crossings input-stream last-value)
  (let ((avpt (/ (+ (head input-stream) last-value) 2)))
    (cons-stream (sign-change-detector avpt last-value)
                 (make-zero-crossings (tail input-stream)
                                      avpt))))
```

This seems to work, but a close look at the output shows that the signals are in fact too smooth! Find the bug that Louis has installed and fix it without changing the structure of the program. Hint: You will have to increase the number of arguments to *make-zero-crossings*.

Exercise 3-51: Eva Lu Ator has a criticism of Louis' approach in exercise 3-50. 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 procedure *smooth* that takes a stream as input and produces a stream, each element of which 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.

Using DELAY to model systems with loops

The *integral* procedure shows how the *delay* built into *cons-stream* enables us to model a system that contains a feedback loop as shown in figure 3-29. Unfortunately, stream models of signal-processing systems with loops may require uses of *delay* beyond the "hidden *delay*" supplied automatically by *cons-stream*. For instance, figure 3-30 shows a signal processing system for solving the differential equation $dy/dt = f(y)$ where f is a given mathematical function. The figure shows a *map* component, which applies f to its input signal, linked in a feedback loop to a integrator, in a manner very similar to analog computer circuits that are actually used to solve such equations.

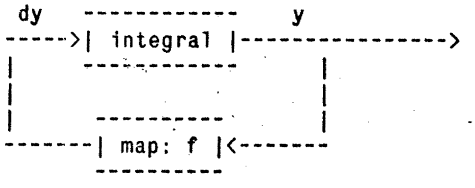


Figure 3-30: An "analog computer circuit" that solves the equation $dy/dt=f(y)$.

Assuming we are given an initial value *y-init* for *y*, we could try to model this system using the procedure:

```
(define (solve f y-init dt)
  (define y (integral dy y-init dt))
  (define dy (map f y))
  y)
```

But this procedure 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*.

> **Exercise 3-52:** Why can't we fix the problem by simply interchanging the first two lines of *solve* to define *dy* before *y*?

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 have properties similar to *cons-stream*, in that we can generate part of the answer given only partial information about the arguments. For *integral*, the head of the output stream is the specified *initial-value*. So we can generate the head of the output stream without evaluating the integrand *dy*. Once we know the *head* of *y*, the *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'll redefine *integral* to expect the integrand stream to be a *delayed* argument. *Integral* will *force* the integrand to be evaluated only when it is required to generate more than the *head* of the output stream:

```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (let ((integrand (force delayed-integrand)))
                   (add-streams (scale-stream dt integrand)
                                int))))
    int)
```

Now we can implement our *solve* procedure by delaying the evaluation of *dy* in the definition of *y*:

```
(define (solve f y-init dt)
  (define y (integral (delay dy) y-init dt))
  (define dy (map f y))
  y)
```

In general, every caller of *integral* must now *delay* the integrand argument.

This explicit use of *delay* provides us with flexibility, but it is awkward. One way around this is to define *integral* to be a special form, similar to *cons-stream*, which will automatically wrap a *delay* around its first argument. This has a disadvantage, though, in that *integral* will no longer be an ordinary procedure, which will cause problems if we wish to use *integral* in conjunction with higher-order procedures. (Compare the problem with *cons-stream* described in exercise 3-40.)

A straightforward way to eliminate this problem once and for all is to make a fundamental change in our programming language, and adopt a model of evaluation in which *all* arguments to procedures 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 Chapter 1 section 1.1.5. This provides a uniform and elegant solution to the problem of using streams to model systems with loops, and it would be a natural strategy to adopt if we were concerned only with stream processing. In section 4.2.2, after we have studied the evaluator, we will see how to transform our language in just this way. Unfortunately, including `delays` in procedure calls wreaks havoc with our ability to understand programs that use assignment. Even the single `delay` in `cons-stream` causes problems, as illustrated by the confusing examples in the exercises at the end of section 3.4.3. As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, a devising ways to deal with both of these at once is an active area of research.

Exercise 3-53: The `integral` procedure used above was analogous to the "implicit" definition of the infinite stream of integers in section 3.4.4. Alternatively, we can give a definition of `integral` that is more like the "generating function" procedure that used `integers-starting-from`:

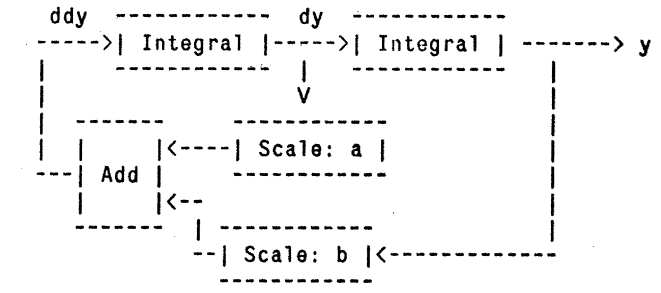
```
(define (integral integrand initial-value dt)
  (cons-stream initial-value
               (if (empty-stream? integrand)
                   the-empty-stream
                   (integral (tail integrand)
                             (+ (* dt (head integrand)) initial-value)
                             dt))))
```

When used in systems with loops, this procedure has the same problem as does our original version of `integral`. Modify the procedure so that it expects the `integrand` as a delayed argument, and hence can be used in the `solve` procedure shown above.

Exercise 3-54: Consider the problem of designing a signal-processing system to study the homogeneous second order linear differential equation

$$d^2y/dt^2 - a\,dy/dt - by = 0$$

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 looks like this:



Write a procedure `2nd`, that takes as arguments the constants `a`, `b`, and `dt`, together with initial values `y-init` and `dy-init` for y and dy/dt, and generates the stream of successive values of y.

Exercise 3-55: Generalize the `2nd` procedure of exercise 3-54 so that it can be used to solve general second order differential equations $y = f(dy/dt, d^2y/dt^2)$.

Exercise 3-56: A *series RLC-circuit* consists of a resistor, a capacitor, and an inductor connected in series. If R, L, and C measure the resistance, inductance, and capacitance, V_C is the voltage across the capacitor, and I_L the current in the inductor, then the state of the circuit is determined by the pair of

differential equations:

$$\frac{dV_C}{dt} = -I_L/C$$

$$\frac{dI_L}{dt} = 1/L\ V_C - R/L\ I_L$$

The signal-flow diagram representing this system of differential equations is shown in figure 3-31.

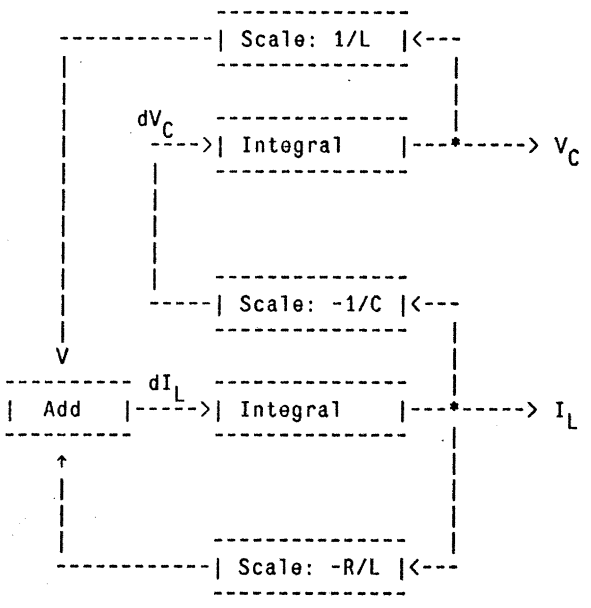


Figure 3-31: A signal-flow diagram for the solution to a series RLC circuit.

Write a procedure, *RLC*, which takes as arguments the device parameters of the circuit, *R*, *L*, and *C*, and the time increment *dt*. In a manner similar to the *RC* procedure of exercise 3-48, *RLC* should produce a procedure which takes the initial values of the state variables, V_{C0} and I_{L0}, and produces a pair (*cons*) of the streams of states, *Vc* and *IL*. Using *RLC*, generate the pair of streams which models the behavior of a series RLC circuit with *R* = 1 Ohm, *C* = 0.2 Farad, *L* = 1 Henry, and *dt* = 0.1 second, and with initial values of the state variables, I_{L0} = 0 Amps, and V_{C0} = 10 Volts.

3.4.6. Using Streams to Model Local State

Let us now return to the beginning of this chapter on objects and state and examine it in a new light. We introduced assignment statements and mutable objects in an attempt to improve the modularity of programs that model systems with local 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 corresponding computational objects. 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 the successive values. 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 in the computer during evaluation. Indeed, due to the presence of *delay*, there may be little relation between

simulated time in the model and the order of events in the computer.

In order to contrast these two approaches to modeling, let us return to the problem of implementing a "withdrawal processor" that monitors the balance in a bank account. In section 3.1.2 we implemented a simplified version of such a processor:

```
(define (make-simplified-withdraw balance)
    (lambda (amount)
      (set! balance (- balance amount))
      balance))
```

Calls to *make-simplified-withdraw* produce computational objects, each with a local state variable *balance*, which is decremented by each successive call 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 successive inputs to such an object and watching the sequence of returned values shown on a display.

Alternatively, we can model our withdrawal object as a procedure that takes as input a *stream* of amounts and produces the stream of successive balances in the account:

```
(define (stream-withdraw balance amount-stream)
    (cons-stream balance
                 (stream-withdraw (- balance
                                     (head amount-stream))
                                  (tail amount-stream))))
```

Now suppose 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 point of the user who is typing values and watching results, the stream process has *the same behavior* as the *simplified-withdraw* object. And yet, with the stream version, there is no assignment, no local state variable, and consequently none of the theoretical problems that we encountered in section 3.1.2. The system appears to have state, and yet, somehow, there is no state![38]

Exercise 3-57: Extend the *stream-withdraw* procedure to a more complete model for bank accounts, thus producing a stream analogue of the *make-account* procedure of section 3.1.1.

Monte Carlo simulation, revisited

As we saw in section 3.1.3, 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 re-implement the Monte Carlo estimation of π, which we examined in section 3.1.3, from a stream-processing point of view.

Recall that the key modularity issue was that we wished to hide the internal state of a

---

[38] This is really remarkable. Even though *stream-withdraw* is a well-defined mathematical function, whose behavior does not change, the user's perception here is that he is 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. For example, 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. Similarly, if the user could step back from the interaction and think in terms of streams of balances rather than individual transactions, he could regard the system as stateless.

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:

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

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*:

```
(define random-numbers
  (cons-stream random-init
               (map rand-update random-numbers)))
```

We use this to construct the stream of outcomes of the *cesaro* experiment performed on consecutive pairs in the *random-numbers* stream.

```
(define cesaro-stream
  (map-successive-pairs (lambda (r1 r2) (= (gcd r1 r2) 1))
                        random-numbers))

(define (map-successive-pairs f s)
  (cons-stream (f (head s) (head (tail s)))
               (map-successive-pairs f (tail (tail s)))))
```

The *cesaro-stream* is now fed to a *monte-carlo* procedure, which produces a stream of estimates of probabilities. The results are then converted into a stream of estimates of π.

```
(define (monte-carlo experiment-stream nt nf)
  (define (next nt nf)
    (cons-stream (/ nt (+ nt nf))
                 (monte-carlo (tail experiment-stream) nt nf)))
  (if (head experiment-stream)
      (next (+ nt 1) nf)
      (next nt (+ nf 1))))

(define pi
  (map (lambda (p) (sqrt (/ 6 p)))
       (monte-carlo cesaro-stream 0 0)))
```

There is considerable modularity in this approach, because we still can formulate a general *monte-carlo* procedure that can deal with arbitrary experiments. Yet there is no assignment or local state.

Exercise 3-58: Exercise 3-7 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, which operates on an input stream of requests to *generate* a new random number or to *reset* the sequence to a specified value, and which produces the desired stream of random numbers.

Streams versus objects

Streams and delayed evaluation can be powerful modeling tools, providing many of the benefits of local state and assignment. Moreover, they avoid the theoretical problems which, as we saw in section 3.1.2, must accompany the introduction of assignment into a programming language. This observation has led many current researchers to propose so-called *functional programming languages*, which make heavy use of delayed evaluation, but do not include any provision for assignment or mutable data. The functional approach is also extremely attractive when we consider the problem of designing programming languages for *multiprocessing computers*, in which many computations are carried out in parallel. The absence of assignment means that the programmer need not be concerned with synchronization errors caused by values being updated in the wrong order, and the possibility of such errors poses a major problem when implementing traditional programs on multiprocessing systems. Because of this, it seems certain that functional methods will play an increasingly important role in the future development of programming languages and techniques.[39]

On the other hand, it is an open question whether *all* need for assignment can be reasonably bypassed using delayed evaluation. One particularly troublesome area arises when we wish to design interactive systems, particularly ones that model interactions between truly independent entities. For instance, consider the problem of implementing a banking system that permits joint bank accounts. In a conventional system using assignment and objects, we would model the fact that Peter and Paul share an account by having both Peter and Paul send their transaction requests to the same bank account object, as we saw in section 3.1.2. From the stream point of view, where there are no "objects" *per se*, we have already indicated (exercise 3-57) 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 shown in figure 3-32.

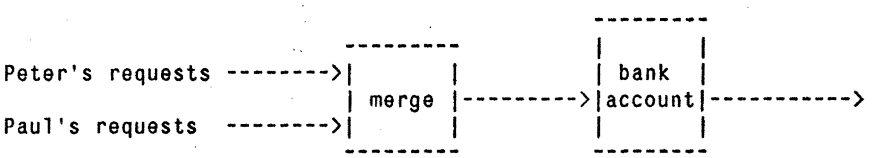


Figure 3-32: A joint bank account, modeled by merging two streams of transaction requests.

The problem 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. For 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! In order to deal with

---

[39] John Backus, the inventor of FORTRAN, gave high visibility to the functional programming movement when he was awarded the ACM Turing award in 1978, giving an acceptance speech [2] that strongly advocated the functional approach. A good overview of functional programming is given in the books by Henderson [19] and by Darlington, Henderson, and Turner [9].

this problem, most researchers agree that it is necessary to supplement the purely functional basis of the language with some new constructs. One of these, known as *fair merge* (or *non-deterministic merge*) is just what we need for the joint account problem. Roughly, the fair merge of two streams will wait for an input to appear on either stream and produce the received value, alternating between the two streams in some fair way whenever both of them produce inputs. Notice that even the definition of fair merge involves introducing the notion of time (through waiting) which is precisely what we avoid when we adopt the functional approach. Extensions such as fair merge are troublesome from a theoretical point of view, because they are not well-understood, and because adding too powerful a construct could well enable us to implement assignment, thus re-introducing many of the same problems that the functional style was meant to guard against in the first place.[40]

Another problem with the stream formulation is that it seems inherently biased towards models whose components have "inputs" and "outputs." For example, one can use streams as the basis for an elegant reformulation of the digital circuit simulator of section 3.3.4, in which the elementary gates are viewed as processes on streams of 0's and 1's. On the other hand, it is unclear how to give a natural stream formulation of the constraint propagation system of section 3.3.5, where in maintaining a constraint such as $A+B=C$, there is no fixed element that can be viewed as the output determined by the other two values. In general, it seems that some problems are more naturally viewed in terms of clusters of communicating entities than in terms of signal-flow. Perhaps the best that one can say at present is that "objects" and streams both lead to powerful modeling disciplines. The choice between them is far from clear, and the search for a uniform approach that combines the benefits of both of these perspectives is a central concern of current research in programming methodology.

---

[40] Even fair merge does not solve all our problems. Although we can model shared bank accounts, it is awkward to use fair merge in more complex resource allocation problems. The resource *manager*, introduced by Arvind and Brock [1], is a more powerful construct developed to deal with these more complex situations.

Chapter 4
Meta-Linguistic 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 by using the same general techniques as do 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 Lisp 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 Lisp, 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; for 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.

For example, electrical engineers use many different languages for describing circuits. Two of these are the language of electrical *networks* and the language of 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 elements such as resistors, capacitors, inductors, and transistors, which are characterized in terms of physical variables called voltage and current. In contrast, the electrical system language is concerned with the functional behavior of signal-processing modules, and signals are manipulated without concern for their physical representation as voltages and currents. When describing circuits in the network language, the engineer is concerned with the physical feasibility of a design, for example, with optimizing the gain-bandwidth product of an amplifier by choosing appropriate components. In contrast, the electrical system language emphasizes the teleological properties of electrical devices. It is erected on the network language, in the sense that the entities of signal processing systems are constructed from electrical networks, but the concerns here are the large-scale organization of electrical devices to solve a given application problem, assuming the feasibility of the parts.

Programming is similarly 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 are erected on a machine language substrate in much the same way that the electrical systems language is erected on a substrate of electrical networks. High-level languages hide concerns about the representation of data as collections of bits and the representation of programs as sequences of primitive instructions. Such languages have means of combination and abstraction, such as procedure definition, appropriate to the larger scale organization of systems.

Meta-linguistic abstraction -- establishing new descriptive languages -- plays an important role in all branches of engineering design. But it is particularly important to computer programming because, in programming, not only can we formulate new languages, but we can also *implement* these language by constructing *evaluators*. An evaluator (or interpreter) for a programming language is a procedure which, when applied to an expression of the language, performs the actions required to execute that expression (regarded as a fragment of a program written in the language). Thus, given a language which a computer knows how to execute, if we can implement in that language an evaluator for a second language, then our computer will also be able to execute expressions of the latter language. This method of implementing a language is known as constructing an *embedded language*.

In point of fact, it is not too much of a distortion to regard almost any program as the evaluator for some language. For instance, the polynomial manipulation system of section 2.4.3 embodies the rules of polynomial arithmetic and implements them in terms of computer operations on list structured data. If we augment this system with procedures 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 ultimately 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.

Embedded 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 Lisp as a base, implementing evaluators as Lisp procedures. Lisp is particularly well-suited to this task, due to its ability to represent and to manipulate symbolic expressions. We will take the first step in understanding how languages are implemented by building an evaluator for Lisp itself. The language implemented by our evaluator will be a subset the Scheme dialect of Lisp that we have been using in this book. Although the evaluator described in this chapter is written for a particular dialect of Lisp, 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 "Lisp" evaluator. The evaluator has been simplified for the purposes of illustration and discussion, and there are a number of features that have been left

out which would be important to include in a production quality Lisp system.[1] Nevertheless, this simple evaluator is adequate to execute most of the programs in this book.

Our evaluator for Lisp will be implemented as a Lisp program. It may seem circular, to think about evaluating Lisp programs using an evaluator that is itself implemented in Lisp. Indeed, an evaluator that can execute itself is said to be *meta-circular*. We will use our Lisp meta-circular evaluator to help us to understand the basic structure of any evaluator. In essence, evaluation is a process, and we will be using the Lisp programming language -- which is our tool for describing processes -- to describe the evaluation process itself.[2]

An important advantage of formulating the evaluator as a Lisp program is that we can consider alternative evaluation rules by describing these as modifications to the evaluator program. We will use this technique in section 4.2 to explore variations of the Scheme dialect of Lisp, including a language in which variables are scoped dynamically, and a language which uses normal rather than applicative order evaluation. Both of these can be implemented by making modest changes to the original Scheme evaluator. The final section of this chapter presents an extended example of meta-linguistic abstraction. We implement a *logic programming* language, which allows a programmer to retrieve information from data bases by formulating *queries* and *rules of inference*. Even though the query language is strikingly different from Lisp (or any other procedural language) we will discover that the evaluator for the query language contains many of the central elements found in a Lisp evaluator.

4.1. The Meta-circular Evaluator

We will implement our evaluator as a procedure *eval* of two arguments -- an expression together with an environment in which the expression is to be evaluated. *Eval* will embody, and concretize, the environment model of evaluation that we described in Chapter 3, section 3.2. Recall that the model has two basic parts. The first part specifies that to evaluate a compound expression (other than a special form) we

- Evaluate the subexpressions, and then apply the value of the operator subexpression to the values of the operand subexpressions.

The second part of the evaluation model specifies that to apply a compound procedure to a set of arguments we

- Evaluate the body of the procedure in a new environment. This environment is constructed by extending the environment part of the procedure object, by a frame in which the formal parameters of the procedure are bound to the actual

---

[1] The most important of these are mechanisms for handling errors and supporting debugging.

[2] Even so, there will remain important aspects of the evaluation process that are not elucidated by the meta-circular evaluator. Most important of these are the detailed mechanisms by which procedures call other procedures and return values to their callers. We will address these issues in Chapter 5, when we take a closer look at the evaluation process by transforming the meta-circular evaluator into a program that runs on a very simple machine.

arguments to which the procedure is to be applied.

These two rules describe the essence of the evaluation process, a basic cycle in which expressions to be evaluated in environments are reduced to procedures to be applied to arguments, which in turn are reduced to new expressions to be evaluated in new environments, and so on, until expressions are reduced to symbols, whose values are looked up in the environment, and to primitive procedures, which are applied directly. This evaluation cycle will be embodied by the interplay between the two critical procedures in the evaluator, *eval* and *apply*.

To model the application of primitive procedures, we assume that we have available a Lisp procedure called *apply-primitive-procedure* that takes as arguments a primitive procedure and a set of values and returns the result of applying the procedure to the values as arguments.[3]

In addition to the ability to apply primitives, we need operations for manipulating environments. As explained in section 3.2, an environment is a sequence of frames, where each frame is a table bindings which pair variables with their corresponding values. We assume that we have available the following operations for manipulating environments.

(lookup-variable-value variable env)

Takes a variable and an environment as arguments. It returns the value bound to the variable in the environment, or signals an error if the variable is unbound.

(extend-environment variables values base-env)

Takes a list of variables, a list of values, and an environment as arguments. It returns a new environment, consisting of a frame in which the variables are bound to the corresponding values, whose enclosing environment is the specified *base-environment*.

---

[3] One might ask: If we grant ourselves the ability to apply primitives, then what remains for us to implement in the evaluator? The answer to this question is that the evaluator performs three essential tasks, and it is precisely these three things that make a language more than merely a collection of primitive operators:

1. The evaluator enables us to deal with *compound expressions*. For example, while the simple mechanism of *apply-primitive-procedure* would suffice for evaluating the expression *(+ 1 6)*, it is not able to handle *(+ 1 (* 2 3))*, because as far as the primitive procedure + 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 the composition of functions so that *(* 2 3)* is reduced to 6 before being in turn passed as an argument to +.
2. The evaluator allows us to use *variables*. For example, the primitive procedure + has no way to deal with expressions such as *(+ x 1)*. We need an evaluator to keep track of variables and replace them with their values before invoking the primitive procedures.
3. The evaluator allows us to define *compound procedures*. This involves keeping track of procedure definitions, knowing how to use these definitions in evaluating expressions, and providing a mechanism that enables procedures to accept parameters.

In other words: *the job of the evaluator is not so much 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.*

(define-variable! variable value env)
Adds a new binding to the first frame in the environment. The new binding associates the specified variable with the specified value.

(set-variable-value! variable value env)
Changes the binding of the variable in the environment, so that the variable is now bound to the specified value. It signals an error if the variable is unbound.

In adherence to the discipline of data abstraction, our evaluator will operate on environments environments using these procedures, without making any commitment to how environments are represented. The representation of environments is discussed separately, in section 4.1.3.

4.1.1. The core of the evaluator: EVAL and APPLY

There are two key procedures in the evaluator, called *eval* and *apply*. The *eval* procedure takes as arguments an expression and an environment. *Eval* classifies the expression and directs its evaluation. In the case of an ordinary procedure application, this involves finding the procedure and the evaluated arguments and setting things up for the application. *Apply* handles the actual procedure application itself, both for primitive procedures and for compound procedures. For primitive procedures this is accomplished by calling *apply-primitive-procedure*. For compound procedures, *apply* must perform the process dictated by the environment model of evaluation. Namely, it must construct a new environment, and, in that environment, evaluate the procedure body. This entails calling *eval* with the procedure body and the new environment.

EVAL

There are various types of expressions that *eval* must handle:

- For self-evaluating expressions, such as numbers, *eval* returns the expression itself.
- For quoted expressions, *eval* returns the expression that was quoted.
- Variables must be looked up in the environment to find their values.
- An assignment to (or a definition of) a variable, must recursively call *eval* to compute the new value to be associated with the variable. The environment must be modified to change (or create) the binding of the appropriate variable.
- A lambda expression must be transformed into an applicable procedure by attaching an environment to the specified procedure text.
- A conditional expression requires special processing of its clauses.
- Finally, the expression may be an ordinary procedure application. In this case *eval* must determine the procedure referred to by the operator part of the application, and it must evaluate the operands of the application. This recursive evaluation is performed by the auxiliary procedure *list-of-values*, and the result is passed to *apply*.

Eval is structured as a case analysis on the syntactic type of the expression to be evaluated. In order to keep the procedure suitably general, we express the determination of the type of an expression abstractly, making no commitment as to how the various types of expressions are implemented. Each type of expression has a predicate that tests for it, and an abstract means for selecting its parts. This *abstract syntax* makes it easy for us to see how we might use a similar evaluator to interpret another language.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

For clarity, we have implemented *eval* as a case analysis using *cond*. The disadvantage of this is that our procedure handles only a few distinguishable types of expressions, and no new ones can be defined without editing the definition of *eval*. In most Lisp implementations, dispatching on the type of an expression is done in a data-directed style. This allows a user to add new types of expressions that *eval* can distinguish, without modifying the definition of *eval* itself.

> **Exercise 4-1:** Rewrite *eval* so that the dispatch is done in data-directed style. You will have to initialize an appropriate table to hold the dispatch procedures. Compare this with the data-directed differentiation procedure of exercise 2-43.

APPLY

The procedure *apply* takes two inputs, a procedure to be executed, and a list of arguments to which the procedure is to be applied. *Apply* classifies procedures into two kinds and directs their application. Primitive procedures are executed directly by *apply-primitive-procedure*. Compound procedures require that the expressions comprising the body of the procedure be sequentially executed (using an auxiliary procedure *eval-sequence*) in an environment where the parameters of the procedure are associated with the arguments passed to the procedure.

Associating the formal parameters of the procedure to the actual arguments is done using the procedure *extend-environment*. *Extend-environment* creates a new environment, which extends a given environment by a new set of bindings. In the case of *apply*, we extend the environment carried by the procedure, by adjoining the association of the formal parameters to the actual arguments.

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                        (extend-environment
                         (parameters procedure)
                         arguments
                         (procedure-environment procedure))))
        (else
         (error "Unknown procedure type -- APPLY" procedure))))
```

Arguments to procedures

When *eval* processes a a procedure application, it uses *list-of-values* to produce the list of arguments to which the procedure is to be applied. *List-of-values* takes as an argument the operands of the combination. It evaluates each operand and returns the list of corresponding values:

```
(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (eval (first-operand exps) env)
                    (list-of-values (rest-operands exps)
                                    env)))))
```

Sequences

The following procedure is used by *apply* to evaluate the sequence of expressions in a procedure body. It takes as arguments a sequence of expressions and an environment, and evaluates the expressions in the order in which they occur. The value returned is the value of the final expression:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

Conditionals

Eval-cond is a procedure that scans the list of clauses of a conditional expression, evaluating the predicate part of each clause to see if it is true. If the predicate part is true, the action sequence part of that clause is executed. If a predicate part is not true, the scan continues. Running out of clauses causes the *cond* to return *nil*.

```
(define (eval-cond clist env)
  (cond ((no-clauses? clist) nil)
        ((else-clause? (first-clause clist))
         (eval-sequence (action-sequence (first-clause clist))
                        env))
        ((true? (eval (predicate (first-clause clist)) env))
         (eval-sequence (action-sequence (first-clause clist))
                        env))
        (else (eval-cond (rest-clauses clist) env))))
```

Assignments to variables

The following procedure handles assignments to variables. It calls *eval* to find the value to be assigned, and transmits the variable and the resulting value to a procedure *set-variable-value!* that will install this information in the designated environment. (See section 4.1.3.)

```
(define (eval-assignment exp env)
  (let ((new-value (eval (assignment-value exp) env)))
    (set-variable-value! (assignment-variable exp)
                         new-value
                         env)
    new-value))
```

Definitions of variables are handled in a similar manner.

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  (definition-variable exp))
```

4.1.2. Representing Expressions

The evaluator is reminiscent of the symbolic differentiation program that we discussed in Chapter 2, section 2.2.4. 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 by combining these 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 procedure could deal with algebraic expressions in prefix form, in infix form, or whatever. For the evaluator, this means that the syntax of the language being evaluated is determined solely by the procedures that classify and extract pieces of expressions.

The following set of procedures defines the syntax of the Lisp dialect used in this book.

- The only self-evaluating items are numbers:[4]

```
(define (self-evaluating? exp) (number? exp))
```

- Quotations are expressions of the form *(quote <text-of-quotation>)*[5]

---

[4] Many Lisp implementations also treat the symbols *nil* and *t* as self-evaluating. In Scheme, *nil* and *t* are ordinary symbols, which are initially bound in the global environment to appropriate values.

[5] As mentioned in Chapter 2, section 2.2.3, this expanded *quote* form is the form in which the evaluator sees quoted expressions, even if these expressions are typed in to Lisp using the quotation mark. For example, the expression *'a* would be seen by the evaluator as *(quote a)*. See also exercise 2-28.

```
(define (quoted? exp)
  (if (atom? exp)
      nil
      (eq? (car exp) 'quote)))

(define (text-of-quotation exp) (cadr exp))
```

- Variables are represented by symbols:

```
(define (variable? exp) (symbol? exp))
```

- Assignments are expressions of the form *(set! <variable> <value>)*

```
(define (assignment? exp)
  (if (atom? exp)
      nil
      (eq? (car exp) 'set!)))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

- Definitions are expressions of the form *(define <variable> <value>)* or of the form

```
(define (<variable> <variable> ... <variable>)
  <body>)
```

The latter form (standard procedure definition) is syntactic sugar for

```
(define <variable>
  (lambda (<variable> ... <variable>)
    <body>))
```

Here are the corresponding syntax procedures:

```
(define (definition? exp)
  (if (atom? exp)
      nil
      (eq? (car exp) 'define)))

(define (definition-variable exp)
  (cond ((variable? (cadr exp))
         (cadr exp))
        (else
         (caadr exp))))

(define (definition-value exp)
  (cond ((variable? (cadr exp))
         (caddr exp))
        (else
         (cons 'lambda
               (cons (cdadr exp)      ;Formal parameters
                     (cddr exp)))))) ;body
```

- Lambda expressions are lists that begin with *lambda*:

```
(define (lambda? exp)
  (if (atom? exp)
      nil
      (eq? (car exp) 'lambda)))
```

- Conditionals begin with *cond* and have a list of predicate-action clauses. A predicate is considered to be true if it is non-*nil*. A clause is an *else* clause if its predicate is the symbol *else*.

```
(define (conditional? exp)
  (if (atom? exp)
      nil
      (eq? (car exp) 'cond)))

(define (clauses exp) (cdr exp))

(define (no-clauses? clauses) (null? clauses))

(define (first-clause clauses) (car clauses))

(define (rest-clauses clauses) (cdr clauses))

(define (predicate clause) (car clause))

(define (action-sequence clause) (cdr clause))

(define (true? x) (not (null? x)))

(define (else-clause? clause)
  (eq? (predicate clause) 'else))
```

- A sequence of expressions is a list of expressions, given in the order in which they are to be evaluated:

```
(define (last-exp? seq) (eq? (cdr seq) nil))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))
```

- A procedure application is any non-atomic expression that is not one of the above expression types. The *car* of the expression is the operator, and the *cdr* is the list of operands:

```
(define (application? exp) (not (atom? exp)))

(define (operator app) (car app))

(define (operands app) (cdr app))

(define (no-operands? args) (eq? args nil))

(define (first-operand args) (car args))
```

```
(define (rest-operands args) (cdr args))
```

- Applicable procedures are constructed from lambda expressions and environments using the constructor *make-procedure*.

```
(define (make-procedure lambda-exp env)
  (list 'procedure lambda-exp env))

(define (compound-procedure? proc)
  (if (atom? proc)
      nil
      (eq? (car proc) 'procedure)))
```

Since a lambda expression has the syntax

(lambda ⟨*parameters*⟩ ⟨exp_1⟩ ... ⟨exp_n⟩)

the result of *make-procedure* is

(procedure (lambda ⟨*parameters*⟩ ⟨exp_1⟩ ... ⟨exp_n⟩) ⟨*env*⟩)

Thus the selectors for the parts of a procedure are:

```
(define (parameters proc) (cadr (cadr proc)))

(define (procedure-body proc) (cddr (cadr proc)))

(define (procedure-environment proc) (caddr proc))
```

Exercise 4-2: The interpreter described above supports *cond* but not *if*. Modify the interpreter to add *if* to the language. Following the style used in the rest of the implementation, you should define selectors that return the various parts of an *if* statement, and a procedure *eval-if* that is analogous to *eval-cond*.

Exercise 4-3: The *let* statement is simply syntactic sugar because

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

is equivalent to

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

Modify the evaluator to recognize and correctly handle *let* statements.

Exercise 4-4: By using of data abstraction, we were able to write an *eval* procedure that is independent of the particular syntax of the language to be evaluated. To illustrate this, design and implement a new syntax for Lisp by modifying the procedures in this section.

4.1.3. Representing environments

An environment is structured as a sequence of frames, where each frame is a table that associates variables with values. We can implement a frame as a list of *bindings*, where each binding pairs a variable with its associated value. If we represent bindings as Lisp pairs, then we have the following procedures for constructing bindings, selecting the variable and value parts of a binding, and changing the value part of a specified binding:

```
(define (make-binding variable value)
  (cons variable value))

(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (set-binding-value! binding value)
  (set-cdr! binding value))
```

Since a frame is a list of bindings (pairs), we can get the binding of a variable in a given frame using the *assq* operation, which was described in section 3.3.3.

```
(define (assq key pairs)
  (cond ((null? pairs) nil)
        ((eq? key (caar pairs)) (car pairs))
        (else (assq key (cdr pairs)))))

(define (binding-in-frame var frame)
  (assq var frame))
```

The following procedure takes a list of variables and a list of values as arguments and constructs the corresponding frame. It signals an error if the number of variables is not equal to the number of values:

```
(define (make-frame variables values)
  (cond ((and (null? variables) (null? values)) '())
        ((null? variables)
         (error "Too many values supplied" values))
        ((null? values)
         (error "Too few values supplied" variables))
        (else
         (cons (make-binding (car variables)
                             (car values))
               (make-frame (cdr variables)
                           (cdr values))))))
```

An environment can be represented as a list of frames, in which case we have the following operations for selecting the first frame in a given environment, selecting all but the first frame, adjoining a frame to an environment, and changing the first frame of an environment:

```
(define (first-frame env) (car env))

(define (rest-frames env) (cdr env))
```

```
(define (adjoin-frame frame env) (cons frame env))

(define (set-first-frame! env new-frame)
  (set-car! env new-frame))
```

The following procedure returns the binding of a variable in a specified environment. It searches each frame in succession, until it finds a frame in which the variable has a non-*null* binding:

```
(define (binding-in-env var env)
  (if (null? env)
      nil
      (let ((b (binding-in-frame var
                                 (first-frame env))))
        (if (not (null? b))
            b
            (binding-in-env var (rest-frames env))))))
```

The operations on environments that are required by the evaluator can now be implemented using the above procedures as building-blocks. Looking up variable in an environment is accomplished by finding the binding for the variable and returning the value part of the binding. An error is signalled if no binding is found.

```
(define (lookup-variable-value var env)
  (let ((b (binding-in-env var env)))
    (if (null? b)
        (error "Unbound variable" var)
        (binding-value b))))
```

To extend an environment by a new frame that associates variables to values, we construct a new frame of bindings and adjoin this frame to the environment:

```
(define (extend-environment variables values base-env)
  (adjoin-frame (make-frame variables values) base-env))
```

To set a variable to new value in a specified environment, we alter the value part of the binding, or else signal an error if the variable is unbound.

```
(define (set-variable-value! var val env)
  (let ((b (binding-in-env var env)))
    (if (null? b)
        (error "Unbound variable" var)
        (set-binding-value! b val))))
```

To define a variable, we insert a new binding pair at the head of the first frame in the environment.

```
(define (define-variable! var val env)
  (set-first-frame! env
                    (cons (make-binding var val)
                          (first-frame env))))
```

As is usually the case with data structures, there are many plausible alternative ways to reprepresent environments, and we can isolate the rest of the evaluator from the detailed choice of representation by taking advantage of data abstraction. In a production Lisp

system, the speed of evaluator's environment operations, especially 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. There are many other ways to implement environments, the best of which allow fast access to variables when running compiled code.[6]

Exercise 4-5: Instead of representing a frame as a list of pairs, you can represent a frame as a pair of lists, a list of names and a list of corresponding values. This speeds up procedure application, since *make-frame* is now implemented simply as *cons* (if we ignore error checking). Rewrite the other environment operations to use this new representation.

Exercise 4-6: Lisp allows us to define new symbols by means of *define*, but provides no way to get rid of symbols. Implement for the interpreter an operation *make-unbound*, which takes a symbol as argument and removes its binding from the current environment.

4.1.4. Running the Evaluator as a Lisp Program

This completes the definition of the evaluator. We have used Lisp -- our chosen language for describing processes -- to describe the process by which Lisp expressions themselves are evaluated. In fact, we can actually run the evaluator as a Lisp program. This will provide a working "Lisp within Lisp," which can serve as a framework for experimenting with alternative evaluation rules, as we shall do in section 4.2.

If we want to run the evaluator, we need a mechanism for applying primitive procedures. We must create an appropriate object corresponding to each primitive procedure we wish to include in the language. It does not matter what these primitive objects are, so long as they can be identified and applied by the procedures *primitive-procedure?* and *apply-primitive-procedure*. For example, we can use the symbols *primitive-car*, *primitive-cdr*, and so on to represent the primitive operators. If we do this, then we can test to see whether an object is a primitive operator by seeing if it is included in a given list of primitives.

```
(define primitive-procedures
  '(primitive-car primitive-cdr primitive-cons
    ...
    <more primitives>
    ...))

(define (primitive-procedure? p)
  (memq p primitive-procedures))
```

To apply a primitive procedure, we check to see which primitive it is, and perform the application using the underlying Lisp in which the evaluator is implemented.

---

[6] The representation described above is called *deep binding*. Its drawback is that in a deeply nested environment, the evaluator may have to search through many frames in order to find the binding for a given variable. One way to avoid this inefficiency is to make use of a strategy called called *lexical addressing*. We will discuss this in Chapter 5, section 5.4.5.

```
(define (apply-primitive-procedure p args)
  (cond ((eq? p 'primitive-car) (car (car args)))
        ((eq? p 'primitive-cdr) (cdr (car args)))
        ((eq? p 'primitive-cons) (cons (car args)
                                       (cadr args)))
        <and so on>
        ...))
```

Observe that the *args* argument to this procedure is a list of arguments to which the primitive should be applied. Thus, to apply *car* we take the first item in the list -- *(car args)* -- and apply *car*.[7]

Next, we need to set up an appropriate global environment, which associates the primitive objects with the names used to refer to these objects in the expressions that we will be evaluating. The global environment should also include bindings for the symbols **t** and ***nil***.

```
(define primitive-procedure-names
  '(car cdr cons
    <names of more primitives>
    ...))

(define (setup-environment)
  (define initial-env
    (extend-environment primitive-procedure-names
                        primitive-procedures
                        nil))
  (define-variable! 'nil nil initial-env)
  (define-variable 't (not nil) initial-env)
  initial-env)

(define the-global-environment (setup-environment))
```

Exercise 4-7: The above implementations of *primitive-procedure?* and *apply-primitive-procedure*, and construction of the initial global environment are very awkward, because they depend on coordinating the two lists *primitive-procedures* and *primitive-procedure-names*, and also keeping these consistent with the actual underlying operations used in *apply-primitive-procedure*. This will be a likely source of bugs if we begin to add new primitives to our language. Redesign the implementation so that all information is kept in a single, conveniently modified data structure, from which the global environment is constructed at system start-up time. Also rewrite *apply-primitive-procedure* so that it will select the appropriate underlying operation by performing a data-directed dispatch based on information included in the data structure, and thus need not be modified when new primitive procedures are added to the language.

Finally, we provide a *driver-loop* that repeatedly prints a prompt, reads an input expression, evaluates this in the global environment, and prints the result.

---

[7] The interface shown here to the primitives of the underlying Lisp, although straightforward, is extremely cumbersome. Data-directed programming provides a better programming organization. See exercises 4-7 and 4-8.

```
(define (driver-loop)
  (newline)
  (princ "**==>")
  (print (eval (read) the-global-environment))
  (driver-loop))
```

We have chosen the prompt **==> to be distinct from the ordinary system prompt, so that, when we run the interpreter, we can tell whether we are typing at our interpreter, or typing at the underlying Lisp system.

4.1.5. EVAL: Treating Expressions as Programs

In the remainder of this chapter, we shall use our meta-circular evaluator as a tool for exploring some alternative Lisp-like languages. But first, let us ponder the significance of a Lisp program that evaluates Lisp expressions. In thinking about this, 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 following program to compute factorials:

```
(define (fact n)
  (if (= n 0)
      1
      (* (fact (-1+ n)) n)))
```

We may regard this program as the description a machine that contains a decrementer, a multiplier, a zero tester, a two-position switch, and another factorial machine. (The factorial machine is infinite because it contains another factorial machine within it.) Figure 4-1 is a flow diagram for the factorial machine, showing how the parts are wired together.

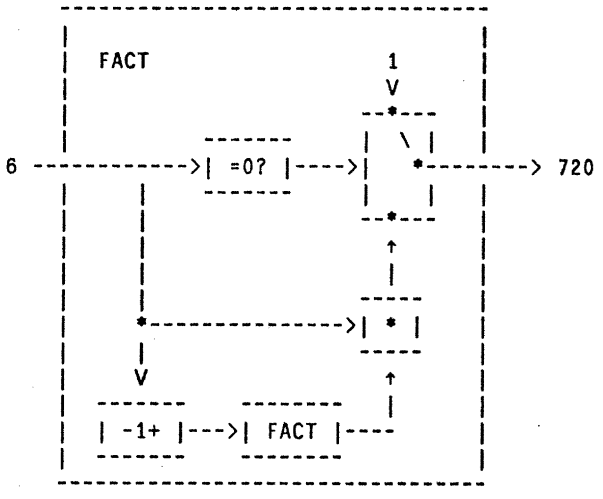


Figure 4-1: The factorial program, viewed as an abstract machine.

In a similar way, we can regard our evaluator as a very special machine, which 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 4-2, then the evaluator will be able to compute factorials.

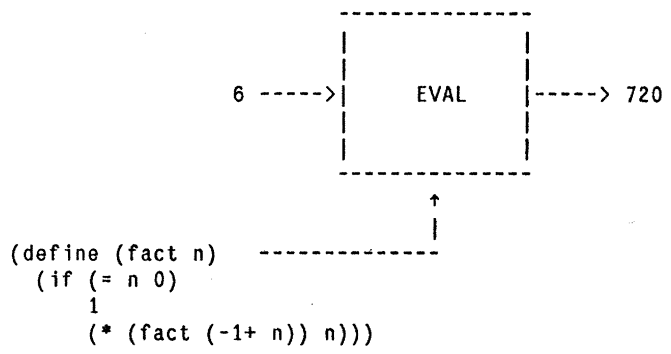


Figure 4-2: The evaluator, simulating a factorial machine.

From this perspective, an evaluator is seen to be a universal mimic machine. It simulates other machines when these are described as Lisp programs. This is quite striking: Try to imagine the analog of an evaluator for electrical circuits. This would be a circuit that takes as input a signal encoding a schematic diagram -- 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 mimic is almost unimaginably complex. It is remarkable that the program evaluator is, in contrast, itself described by a rather simple program.[8]

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 Lisp) 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 is to execute. On the other hand, from the perspective of the evaluator, the expression is simply a list (in this case, a list of three symbols: ***, *x*, and *x*) which 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 data object as a Lisp expression, by making *eval* available for use in programs.

Lisp provides a primitive *eval* procedure, which takes as arguments an expression and an environment, and evaluates the expression in the environment. Thus,

```
(eval '(* 5 5) user-initial-environment)
```

and

---

[8] Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple procedure, can simulate programs that are more complex than the evaluator itself. We are bordering here on deep questions in mathematical logic and computation concerning issues of computability and universal machines. The beautiful book by Hofstadter [23] explores some of these ideas.

```
(eval (cons '* (list 5 5)) user-initial-environment)
```

will both return 25.[9]

In addition to *eval*, Lisp systems also provide *apply* as a user-accessible operation. For example,

```
(apply + (list 1 2 3 4))
```

would return 10. *Apply* is useful in implementing embedded languages, since it provides a uniform way to access the primitives of the underlying Lisp and to incorporate these into an embedded language, as illustrated in exercise 4-8.

Exercise 4-8: Redesign the interface between the meta-evaluator and the underlying Lisp given in section 4.1.4 so that arbitrary Lisp procedures can easily be installed as primitives in the underlying language. You should define a procedure called *install-primitive* which is used as follows:

```
(install-primitive 'square (lambda (x) (* x x)))
```

This installs a primitive called *square* in the embedded Lisp, by binding it in the global environment to an object that consists of the procedure created (in the underlying Lisp) by the *lambda*, together with a tag that allows this to be recognized by *primitive-procedure?* as a primitive procedure. If this is done, then *apply-primitive-procedure* can simply apply the procedure directly by calling *apply* from the underlying Lisp.[10]

---

[9] Warning: The *eval* primitive is not identical to the *eval* procedure that we implemented in section 4.1, because it uses the *actual* Scheme environments, rather than the sample environment structures that we built in section 4.1.3. These actual environments can *not* be manipulated by user as ordinary lists, and must be accessed via *eval* or other special operations. In the MIT implementation of Scheme, *user-initial-environment* is a symbol that is bound to the initial environment in which inputs are evaluated when the system is started up. There is also a primitive procedure called *the-environment* which returns the current environment. Exploiting the ability to access environments can lead to great programming power. (We will see examples of how to use environments to package information in section 5.1.5 and in Appendix I.) However, as with most powerful mechanisms, manipulating environments must be performed with care and respect.

[10] One technical problem you will encounter here is that the *apply* you must call is the primitive operator *apply*, while the meta-circular evaluator defines its own procedure *apply* that will mask the definition of the primitive. One way around this is to rename *eval* and *apply* to avoid conflicts with the names of primitive operators. A more elegant solution, available in the MIT implementation of Scheme, is to use

```
(define apply-in-underlying-scheme (access apply nil))
```

Access is a special form that looks up the binding of a given symbol in a given environment. In this case, we are getting the value of *apply* from the system global environment. This is an example of how the explicit use of environments can be used to deal with the problems of name conflicts in large programs. See a Appendix I.

4.2. Variations on a Scheme

Now that we have an evaluator expressed as a Lisp program, we can begin to experiment with alternative choices in language design, by simply modifying the evaluator. Indeed, new languages are often invented by first writing an evaluator which embeds the new language within an existing high level language. Not only does this allow the evaluator to be more easily tested and debugged, but the embedding enables the designer to snarf[11] features from the underlying language, just as our embedded Lisp interpreter used primitives and control structure from the underlying Lisp.[12] Only later (if ever) need the designer go to the additional effort to build a complete implementation in a low-level language or in hardware.

In this section, we explore two variations on the Scheme dialect of Lisp. In the first variation, we look at a language in which variables are scoped *dynamically* rather than *statically*. In the second variation, we implement procedures with *call-by-name* parameters so that delayed evaluation is performed automatically. Both of these changes are accomplished with minimal changes to our metacircular evaluator. In section 4.3, we will turn to a much more substantial example of an embedded language.

4.2.1. Alternative Binding Disciplines

In Scheme, free variables in procedures are bound *statically*. A free variable in a procedure gets its value from the environment in which the procedure is defined. This means that the binding of a variable in a program is determined by the static structure of the program, not by its run-time behavior. In this discipline, an occurrence of a variable in an expression always refers to the innermost lexically apparent binding of that variable. For this reason, static binding is also called *lexical scoping*.

For example, recall the *sum* procedure from section 1.3.1 of Chapter 1:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

Sum is a simple example of a procedure that takes a procedure as an argument. It provides a template, capturing the structure of a class of procedures. The procedural argument allows the programmer to fill in the template, tailoring it to his needs. Using *sum* we can define a procedure *sum-cubes*, as follows;

```
(define (cube x)
  (expt x 3))
```

---

[11] SNARF v. To grab, esp. a large document or file for the purpose of using it either with or without the author's permission. See BLT. Variant: SNARF (IT) DOWN.

The above definition was snarfed from "MC:GLS;JARGON >" by Guy L. Steele Jr., *et. al.* [42].

[12] This is very significant in saving work for the designer. It will cost us considerable effort in Chapter 5 building the control structure that we avoided implementing in section 4.1 because we were able to snarf it from the underlying Lisp.

```
(define (sum-cubes a b)
  (sum cube a 1+ b))
```

Surely, we could extend this idea to arbitrary powers by defining a procedure *sum-powers* that takes an argument *n*, specifying the power to which each summand should be raised:

```
(define (sum-powers a b n)
  (define (nth-power x)
    (expt x n))
  (sum nth-power a 1+ b))
```

Note that the definition of *nth-power* is internal to *sum-powers*, so that the *n* which is free in *nth-power* will be in the scope of the formal parameter *n* of *sum-powers*.

This demonstrates the real power of internal definitions. They allow us to notate a procedure with more parametric control than just arguments. We then have two paths to influence its computation. We can communicate with a procedure by passing arguments through formal parameters, and we can bind its free variables. This extra degree of freedom is crucial in creating high-order procedural abstractions.

Dynamic binding

Traditionally, Lisp systems have been implemented so that variables are bound *dynamically* rather than statically.[13] In a language with dynamic binding, free variables in a procedure get their values from the environment from which the procedure is called rather than from the environment in which the procedure is defined. Thus, for example, the free *n* in *nth-power* would get whatever value *n* had when *sum* called it. In this example, since *sum* does not rebind *n*, the only definition of *n* is still the one from *sum-powers*, so the effect is the same. If, on the other hand, *sum* bound an *n* of its own (for example, if it used *n* instead of *next* for its third parameter), then when *nth-power* was called, its free *n* would refer to *sum*'s third argument. Not only is this not the value we intended it to have, but in this case it is not even a number! Dynamic binding violates the principle that changing the name of a parameter throughout a procedure definition (in this example, changing *next* to *n* in *sum*) should not change the behavior of the procedure. This is an important modularity problem because then the user of a procedure which takes a procedural parameter must know that the author of the procedure he is using did not name any of his bound variables in a way which might conflict with the free variables occurring in the procedures being passed as arguments.

It may seem obvious from the above example that static binding is the right thing, but there are reasons to prefer dynamic binding. The most important is that the *sum-powers* program above must, in a lexically scoped language, contain the definition of the *nth-power* routine as a local procedure. Thus, if *nth-power* represents a common pattern of usage, its definition will be repeated as a subdefinition in many contexts. This is, itself, a problem of abstraction. It would be nice to be able to move the definition of *nth-power* to a more global context where it can be shared among many procedures. Thus, for example, in a dynamically

---

[13] APL also uses dynamic binding of free variables. It was thought that dynamic binding was better for interpreted languages. Most other languages, such as those descended from ALGOL-60, are lexically scoped and statically bound..

bound Lisp, if we have both *sum* and *product* accumulator procedures, we can define both *sum-powers* and *product-powers* to share the same *nth-power* routine:

```
(define (sum-powers a b n)
  (sum nth-power a 1+ b))

(define (product-powers a b n)
  (product nth-power a 1+ b))

(define (nth-power x)
    (expt x n))
```

It is precisely the attempt to make this work which motivated the development of *dynamic binding* disciplines in traditional Lisp dialects. Unfortunately, dynamic binding must, of necessity, have the problem of symbol conflicts where higher-order procedures capture free variables in procedures passed as arguments.

Implementing dynamic binding

Remarkably enough, modifying our evaluator so that the language it interprets will use dynamic binding rather than static binding requires only a tiny change. When *apply* builds the environment for executing the body of a compound procedure, it extends the evaluation environment of the combination that called for the procedure application, rather than extending the environment of the procedure's definition. This environment must therefore be passed from *eval* to *apply* as an additional argument. The starred lines are the only ones that need to be altered to implement this change:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)
                env))                          ;***
        (else (error "Unknown expression type -- EVAL" exp))))

(define (apply procedure arguments env)   ;***
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                        (extend-environment
                         (parameters procedure)
                         arguments
                         env)))            ;***
        (else
         (error "Unknown procedure type -- APPLY" procedure))))
```

Also, in a dynamically scoped Lisp, it is unnecessary for *make-procedure* to attach the definition environment to a procedure, since this is never used.

Exercise 4-9: Consider the following simple procedure:

```
(define (fact n)
  (cond ((= n 0) 1)
        (else (* (fact (-1+ n)) n))))
```

Suppose that this definition is executed in the global environment. If variables are lexically bound, how many frames with variable *n* must be searched before the value of * is found? What happens if variables are dynamically bound? Draw environment diagrams to illustrate your answer.

Exercise 4-10: Dynamically scoped languages do not conveniently allow a procedure to constrain the values of free variables in a procedure that it returns as a value. Consider the following example:

```
(define (make-adder increment)
  (lambda (x) (+ x increment)))
```

What happens if we attempt to evaluate *((make-adder 3) 4)* in a dynamically bound Lisp?

Exercise 4-11: Despite disadvantages, there are circumstances where dynamically bound variables can be helpful in structuring large programs, since they can simplify procedure calls by acting as implicit parameters. For example, a low-level routine *nprint* called by the system *print* procedure for printing numbers might reference a free variable called *radix* that specifies the base in which the number is to be printed. Procedures that call *nprint*, such as the system *print* operation, should not need to know about this feature. On the other hand, a user might want to temporarily change the *radix*. He could, of course, set *radix* to a new value and later reset it. Yet if *radix* were a dynamic variable, the binding mechanism could accomplish this setting and resetting automatically, in a structured way, for example:

```
(define (print-in-new-radix number (dynamic radix))
  (print number))

(define (print frob)
  ...
  <expressions that involve nprint>
  ...)

(define (nprint number)
  ...
  (dynamic-reference radix)
  ...)
```

Thus, we might wish to build a Lisp system that has *both* static and dynamic variables. One idea is to maintain two separate environments, one for lexical and one for dynamic variables. We used this strategy in the example above. We specified that *radix* is a dynamic variable in the *print-in-new-radix* procedure, and that it was explicitly referenced as a dynamic variable by *nprint*. We declared *radix* to be dynamic by a new syntax in the formal parameter list of *print-in-new-radix*.

Starting with the lexical evaluator of section 4.1, extend it to include dynamic variables of this type. This will require implementing the *dynamic-reference* special form, and the *dynamic* declaration in parameter lists.

Exercise 4-12: The disadvantage of implementing dynamic variables as in exercise 4-10 is that all dynamic variables in the system are effectively global, and hence name conflicts can occur among them. Lexical scoping prevents exactly these conflicts. Another way to achieve the desirable effects of dynamic binding is to use lexical variables only, but to provide a structured means for temporarily changing the value of a variable. For example,

```
(define (with-new-radix new-radix proc)
  (let ((old-radix radix))
    (set! radix new-radix)
    (let ((value (proc)))
      (set! radix old-radix)
      value)))
```

Show how to use *with-new-radix* to define the *print-in-new-radix* procedure of exercise 4-10. Also, modify the evaluator of section 4.1 to include a new piece of syntactic sugar called *fluid-let*, so that *with-new-radix* could be defined as

```
(define (with-new-radix new-radix proc)
  (fluid-let ((radix new-radix))
    (proc)))
```

4.2.2. Example: Delayed evaluation

In ordinary Lisp, when a procedure is called, all the arguments to the procedure are evaluated. This discipline, as we mentioned in Chapter 1, section 1.1.5, is known as *applicative order evaluation*. We also described an alternative evaluation rule, *normal order evaluation*, which delays evaluation of procedure arguments until the last possible moment. A language that does this is said to use *delayed evaluation*, or to pass parameters in a *call-by-name* style.

For example, consider the procedure

```
(define (try a b)
  (cond ((= a 0) 1)
        (else b)))
```

Executing *(try 0 (/ 1 0))* generates an error in Lisp. In a delayed evaluation language, however, there would be no error, and *try* would return *1*, because the argument *b* to *try* would never be evaluated.

In Chapter 3, we introduced a special form called *delay* to provide a restricted kind of delayed evaluation, and made use of this in implementing streams. We could avoid the need for delay by transforming our language so that all parameters are passed by name. On the other hand, delayed evaluation makes it more difficult to deal with mutation and assignment, as we described in section 3.4.5. We can try to maintain the best of both worlds by including both kinds of parameter passing.[14] For instance, we could define *if* in terms of *cond* as follows:

```
(define (if predicate (name consequent) (name alternative))
  (cond (pred consequent)
        (else alternative)))
```

Here, *consequent* and *alternative* are explicitly delayed when passed to *if*, and implicitly forced when they are used.

It is not difficult to modify the interpreter to transform the language to admit call-by-name parameters. Here are some suggestions and hints for doing this:

1. When a procedure application is evaluated, the *name* arguments are not

[14] Algol 60 includes both kinds of parameter passing, known as *call-by-name* and *call-by-value*.

evaluated. Instead, they are transformed to objects called *thunks*.[15] A thunk is essentially a procedure of no arguments, containing as a body the argument that is being passed, together with the current environment, in which the body is to be evaluated if it is needed. Thus you should define a new data object, a ***thunk***, together with appropriate selectors and constructors, and a predicate that tests for an object being a thunk.

2. When *eval* evaluates an application, it evaluates only some of the arguments and constructs thunks for the others. This means that the operator must be extracted first, and the resulting procedure text consulted to determine which parameters are passed by name. (Watch out here for procedures that are passed as parameters by name!)
3. You are supposed to undelay a thunk only at the last possible moment. One last possible moment is when it is passed to a primitive procedure. So you will have to modify *apply-primitive-procedure* so that it undelays any thunks that were passed to it. Note that undelaying is essentially evaluating the thunk body in the thunk environment.

Exercise 4-13: Make the modifications outlined above to transform the interpreter to use delayed evaluation. Test your implementation by evaluating

```
(define (unless predicate (name default-action) (name exception))
  (if (not predicate)
      default-action
      exception))

(define (fact n)
  (unless (= n 0)
          (* n (fact (-1+ n)))
          1))

(fact 4)
```

Exercise 4-14: Consider the following:

```
(define (foo x)
  (cond (x 0)
        (else 1)))

(foo nil)
```

If the interpreter you implemented in exercise 4-13 responds with a 0, you probably forgot a place where it is necessary to undelay thunks. Find it.

Exercise 4-15: In Chapter 3, we said that streams are like lists, except that the second argument to *cons-stream* is delayed. But with call-by-name, streams can be *identical* to lists. The trick is that ***cons*** should not undelay its arguments, as do the other primitive procedures. Modify *apply-primitive-procedure* so that *cons* does not undelay. (If you like, you can do what Scheme *cons-stream* does, and have *cons* undelay its first argument, but not its second.)

Exercise 4-16: The delayed evaluation mechanism outlined above is very inefficient, because each

---

[15] The word "thunk" derives from the implementation of call-by-name in Algol 60. The origin of this name is unknown to the authors, but we have heard that it refers to the sound that the data makes when pushed onto the stack in a running Algol system.

thunk may be undelayed over and over again, each time it is used by a primitive operator. As mentioned in Chapter 3 section 3.4.3, this can be rectified by memo-izing the thunks appropriately. Explain in detail how you would modify your code to perform this optimization.[16]

4.3. Logic Programming

In Chapter 1, we stressed that computer science deals with *imperative*, or "how to" knowledge, as opposed to mathematics, which deals with *declarative*, or "what is" knowledge. Indeed, programming languages require that the programmer express knowledge in a form that indicates the step-by-step methods that he has selected for solving particular problems. On the other hand, *high-level* languages provide, as part of the language implementation, a substantial amount of methodological knowledge, thus allowing the user to avoid worrying about numerous details of how the computation will progress.

Most programming languages, including Lisp, are organized around computing the values of mathematical *functions*. Moreover, expression-oriented languages (such as Lisp, FORTRAN or Algol) capitalize on the pun that an expression which describes the value of a function may be interpreted as a means of computing that value. Because of this, most programming languages are strongly biased toward unidirectional computation -- with well-defined inputs and outputs. Over the past few years, however, people have begun to experiment with a radically different class of programming languages which relax the bias toward unidirectional computation. We have seen one such example in the section 3.3.5 where the objects of computation are arithmetic constraints. Because the direction and order of computation in such a system is not so well defined, the system must provide even more of the detailed "how to" knowledge. This does not mean that the user is released from that burden, however. There are many constraint networks that implement the same set of constraints. The user must choose a suitable network (from the set of mathematically equivalent networks) for a particular computation.

Logic Programming is a growing movement in Computer Science, which encourages an even more extreme relaxation of the view that programming is about constructing algorithms for computing unidirectional functions. It specifically advocates considering the objects of programming to be mathematical *relations*, which have, in general, multiple answers for any

---

[16] This kind of optimization is known as *call-by-need* parameter passing. Call-by-name causes problems in understanding programs with assignments and in controlling the space complexity of programs. But this is innocence itself when compared with the theoretical problems created by call-by-need in the presence of assignments. The excellent article by Clinger [7] attempts to clarify the multiple dimensions of confusion that arise here.

set of inputs.[17]

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 on lists, which takes two lists as arguments and combines their elements to form a single list. In a procedural language such as Lisp, we could define *append* in terms of the basic list constructor *cons*, as follows:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))
```

This procedure can be regarded as a translation into Lisp of the following two rules:

1. For any *x*, *y*, *z*, and *t*,
   - if *x* and *y* *append* to form *z*
   - then *(cons t x)* and *y* *append* to form *(cons t z)*
2. For any list *y*, *(append nil y)* is *y*

Using the procedure, we can answer questions such as:

- Find the *append* of *(a b)* and *(c d)*.

But notice that the same two rules are also sufficient for answering the following sorts of questions:

- Find a list *y* that appends with *(a b)* to produce *(a b c d)*.
- Find all pairs *x* and *y* such that *(append x y)* is *(a b c d)*.

---

[17] 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, which made such a search plausible, was the discovery in the early 1960's of the *unification algorithm* and the *resolution principle* by J.A. Robinson [37]. This was used, for example, by Green and Raphael [13, 12] as the basis for a deductive question-answering system. For a long time, researchers concentrated on algorithms that are guaranteed to find a proof, if such exists. Such algorithms were difficult to control and to direct toward a proof. Hewitt [20], at MIT, recognized the possibility of merging the control structure of a programming language with the operations of a logic manipulation system. A restricted version of his idea was implemented by Sussman, Winograd, and Charniak [44], and was used as the basis of several significant problem-solving programs, such as [51]. At the same time that this was being done, Colmerauer [8], in Marseille, was developing rule-based systems for manipulating natural language. Kowalski [25, 26], in Edinburgh, recognized that such rules could provide the basis for a programming language based on logic. The merging of these latter two strands led to the development of Prolog, which is currently the best-known logic programming language. Thus, according to the French, Prolog was developed at the University of Marseille, while, according to the British, it was developed at the University of Edinburgh. According to people at MIT, Prolog was developed by both of these groups in an attempt to figure out what Hewitt was talking about in his brilliant, but impenetrable Ph.D. thesis.

In a logic programming language, the programmer writes an *append* "procedure" by teaching the computer the two rules about *append* given above. A large amount of "how to" knowledge is provided automatically by the interpreter to allow this single pair of rules to be used to answer all three types of questions about *append*.[18]

Logic programming is currently an extremely active field of research in computer science. 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 to exhibit other undesirable behavior. Most researchers believe that creating a satisfactory (yet not hopelessly inefficient) logic programming implementation will require the use of radically new, massively parallel computer architectures.[19]

In the previous chapter, we explored the technology of implementing interpreters, and described the elements that are basic to an interpreter for a Lisp-like language, or, indeed, to an interpreter for any conventional language. In this chapter, 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 Lisp, we will find it convenient to describe the language using the same general framework that we have followed 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. In terms of implementation, we will see that an interpreter for a logic programming language is considerably more complex than an interpreter for a language such as Lisp. Nevertheless, we will see that our query language interpreter contains many of the same elements found in the interpreter of the previous chapter. In particular, there will be an *eval* part of the interpreter that classifies expressions according to type, and an *apply* part of the interpreter that implements the language's abstraction mechanism, procedures in the case of Lisp, and so-called *rules* in the case of logic programming. We will also see that 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 our query language implementation is that we make substantial use of *stream processing*, which was introduced in Chapter 3.

---

[18] 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 them 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 $y * y = x$.

[19] Logic programming received a big impetus in 1981 when the Japanese government began an ambitious project aimed at building super fast computers that are optimized to run logic programming languages. (The speed of such computers is to be measured in LIPS -- Logical Inferences Per Second -- rather than the usual FLOPS -- FLoating-point Operations Per Second.) It is interesting to note that the only languages the Japanese report considers worth worrying about for computers of the future are Lisp and Prolog. This has proved rather disconcerting to the bulk of the US computer industry, and to the majority of US computer scientists, who seem entrenched in the Pascal-PL/I-Ada camp.

4.3.1. Deductive Information Retrieval

One of the applications at which logic programming really excels is in providing interfaces to data bases for information retrieval. The query language, which we shall implement in this chapter, is designed to be used in this way.

In order to illustrate how the query system works, we will show how it can be used to manage the data base of personnel records for the Itsey Bitsey Machine Corporation, a thriving high-tech company in the greater Boston area. We'll see how 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 the Itsey Bitsey Machine Corporation contains *assertions* about company personnel. Here is the information about Ben Bitdiddle, the resident computer wizard:

```
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
(telephone (Bitdiddle Ben) 491-4371)
(job (Bitdiddle Ben) (computer wizard))
(salary (Bitdiddle Ben) 40000)
```

Observe that each assertion is a list, in this case a triple, and that elements of the triple can themselves be 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 (Hacker Alyssa P) (Cambridge (Massachusetts Avenue) 78))
(telephone (Hacker Alyssa P) 443-8080)
(job (Hacker Alyssa P) (computer programmer))
(salary (Hacker Alyssa P) 35000)
(supervisor (Hacker Alyssa P) (Bitdiddle Ben))

(address (Fect Cy D) (Cambridge (Ames Street) 3))
(telephone (Fect Cy D) 443-0123)
(job (Fect Cy D) (computer programmer))
(salary (Fect Cy D) 32000)
(supervisor (Fect Cy D) (Bitdiddle Ben))

(address (Tweakit Lem E) (Boston (Bay State Road) 22))
(telephone (Tweakit Lem E) 258-4981)
(job (Tweakit Lem E) (computer technician))
(salary (Tweakit Lem E) 15000)
(supervisor (Tweakit Lem E) (Bitdiddle Ben))
```

There is also a programmer trainee, who is supervised by Alyssa:

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(telephone (Reasoner Louis) 735-0157)
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 20000)
(supervisor (Reasoner Louis) (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 description.

Ben is a high-level employee. His supervisor is the company big wheel himself:

```
(supervisor (Bitdiddle Ben) (Warbucks Oliver))
```

```
(address (Warbucks Oliver) (Swellesley (Top Heap Road)))
(telephone (Warbucks Oliver) unlisted)
(job (Warbucks Oliver) (administration big wheel))
(salary (Warbucks Oliver) 100000)
```

Besides the computer division supervised by Ben, the company has an accounting division, consisting of a chief accountant and his assistant:

```
(address (Scrooge Eben) (Weston (Shady Lane) 10))
(telephone (Scrooge Eben) 696-5555)
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 69000)
(supervisor (Scrooge Eben) (Warbucks Oliver))

(address (Cratchet Robert) (Boston (Commonfilth Avenue) 52))
(telephone (Cratchet Robert) 253-1000)
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 12000)
(supervisor (Cratchet Robert) (Scrooge Eben))
```

There is also a secretary for the big wheel:

```
(address (Forrest Rosemary) (Slumerville (Wishbone Terrace) 5))
(telephone (Forrest Rosemary) 491-2131)
(job (Forrest Rosemary) (administration secretary))
(salary (Forrest Rosemary) 15000)
(supervisor (Forrest Rosemary) (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 both of a computer programmer and a computer technician:

```
(can-do-job (computer wizard) (computer programmer))
(can-do-job (computer wizard) (computer technician))
```

A computer programmer could fill in for a trainee:

```
(can-do-job (computer programmer) (computer programmer trainee))
```

Also, as is well known:

```
(can-do-job (administration secretary) (administration big wheel))
```

Formulating simple queries

The query language allows users to retrieve information from the data base by posing *queries* in response to the system's prompt

```
query-->
```

For example, to find all computer programmers, one can say

```
query-->(job ?x (computer programmer))
```

The system will respond by printing the items:

```
(job (Hacker Alyssa P) (computer programmer))
(job (Fect Cy D) (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 atom `job`, the second can be anything, and the third is the literal list `(computer programmer)`. The "anything" that can be the second item in the matching

list is specified by a *pattern variable* `x`. The general form of a pattern variable is a symbol, taken to be the name of the variable, preceded by a colon. We will see below why it is useful to specify names for pattern variables.

The system responds to a query by printing on the terminal the sequence of all entries in the data base that match the specified pattern.

Here are more examples of simple queries:

```
query-->(address ?x ?y)
```

will list all the addresses. This illustrates that a pattern may have more than one variable. The query

```
query-->(job ?x (computer ?type))
```

matches all job entries whose third item is a two-element list whose first item is *computer*, for example

```
(job (Bitdiddle Ben) (computer wizard))
(job (Hacker Alyssa P) (computer programmer))
```

This same pattern does *not* match

```
(job (Reasoner Louis) (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

```
query-->(job ?x (computer . ?type))
```

The use of the period in this pattern is an example of *dot notation*, in which a period followed by a variable in a list expression is used to designate the rest of the list. Thus, the pattern

```
(computer . ?type)
```

matches the data

```
(computer programmer trainee)
```

with *type* as the list *(programmer trainee)*.

Exercise 4-17: Give simple queries that retrieve the following information from the data base:

- all people supervised by Ben Bitdiddle
- the names and jobs of all people in the accounting division
- the names and addresses of all people who live in Slumerville

Compound queries

Simple queries form the primitive operations of the query language. (Although, as we will see below, the implementation of these primitive operations is far from simple, in terms of the primitive operations of present-day computers.) In order to form compound operations, the query language provides means of combination. One of the things 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*.

And can be used as follows to find the addresses of all the computer programmers:

```
query-->(and (job ?person (computer programmer))
             (address ?person ?where))
```

The resulting output is:

```
(and (job (Hacker Alyssa P) (computer programmer))
     (address (Hacker Alyssa P) (Cambridge (Massachusetts Avenue) 78)))

(and (job (Fect Cy D) (computer programmer))
     (address (Fect Cy D) (Cambridge (Ames Street) 3))
```

In general, an *and* query

(and $\langle pat_1 \rangle$ $\langle pat_2 \rangle$ ... $\langle pat_n \rangle$)

will find all sets of values for the pattern variables that simultaneously match all the patterns in the query. It generates a sequence consisting of copies of the original query with the appropriate pattern variables replaced by (or, as one says, *instantiated with*) the appropriate values. Note that *and* here is not the Lisp primitive *and*, but rather an operation built into the query language.

Or is another means for constructing compound queries. For example,

```
query-->(or (supervisor ?x (Bitdiddle Ben))
            (supervisor ?x (Hacker Alyssa P)))
```

will print a stream of copies of the query, with *x* instantiated with all values that make at least one of the patterns match an item in the data base:

```
(or (supervisor (Hacker Alyssa P) (Bitdiddle Ben))
    (supervisor (Hacker Alyssa P) (Hacker Alyssa P)))

(or (supervisor (Fect Cy D) (Bitdiddle Ben))
    (supervisor (Fect Cy D) (Hacker Alyssa P)))

(or (supervisor (Tweakit Lem E) (Bitdiddle Ben))
    (supervisor (Tweakit Lem E) (Hacker Alyssa P)))

(or (supervisor (Reasoner Louis) (Bitdiddle Ben))
    (supervisor (Reasoner Louis) (Hacker Alyssa P)))
```

Not is another way to form compound queries. For example:

```
query-->(and (supervisor ?x (Bitdiddle Ben))
             (not (job ?x (computer programmer))))
```

finds all people supervised by Ben Bitdiddle who are not computer programmers. *Not* can be thought of as a filter that filters the sequence returned by the rest of the pattern, removing all items for which the values of the pattern variables satisfy the *not* clause.

The final combining form is called *lisp-value*. When included as the first element of a pattern, it specifies that the next element is a Lisp predicate to be applied to the rest of the (instantiated) pattern as arguments. For example, to find all people whose salary is greater than 30000 we could query:

```
query-->(and (salary ?person ?amount)
             (lisp-value > ?amount 30000))
```

Exercise 4-18: Formulate compound queries that retrieve the following information:

- the names, addresses and phone numbers of all people who live in Slumerville

- the names of all people who are supervised by Ben Bitdiddle, together with their addresses
- all people whose salary is less than Ben Bitdiddle's, together with their salary and Ben Bitdiddle's salary
- all people who are supervised by someone who is not in the computer division, together with the supervisor's name and job

Rules

In addition to primitive queries and compound queries, the query language also provides *means for abstracting* compound queries. These are given by *rules*. The following rule:

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (lisp-value equal? ?person-1 ?person-2))))
```

Specifies that two people *live-near* each other if they live in the same town. The final *not* clause is used to prevent the rule from saying that all people live near themselves.

The following rule declares that a person is a wheel in the 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>* *<condition>*)

This declares that the *conclusion* should hold for any set of variable values which satisfy the *condition*. We can regard a rule as a way of *abstracting* a compound query (the clauses that make up the *condition*) so that it can be referred to and manipulated as a single unit (the name specified in the *conclusion*).[20]

Once a rule has been defined, we can use it to form still more complex queries. For instance, to find all computer programmers who live near Ben Bitdiddle, we can ask:

```
query-->(and (job ?x (computer programmer))
             (lives-near ?x (Bitdiddle Ben)))
```

As with compound procedures, rules can be used as parts of other rules, 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 4-19: Define a rule which says that person-1 *can replace* person-2 if either person-1 does the

---

[20] We will also interpret a rule specified with a null condition to mean that the rule conclusion is true for any values of the variables.

same job as person-2, or if 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:

- all people who can replace Cy D. Fect
- all people who can replace someone who is being paid more than they are, together with the two salaries

Exercise 4-20: Define a rule that says that a person is a *big-shot* in a division if the person works in the division, but the person does not have a supervisor who works in the division.

Exercise 4-21: By giving the query

```
query-->(lives-near ?person (Hacker Alyssa P))
```

Alyssa P. Hacker is able to find people who live near her, with whom she can carpool to work. On the other hand, when she tries to find all pairs of people who live near each other:

```
query-->(lives-near ?person-1 ?person-2)
```

she notices that each pair of people who live nearby is listed twice, e.g.,

```
(lives-near (Hacker Alyssa P) (Fect Cy D))
(lives-near (Fect Cy D) (Hacker Alyssa P))
.
.
.
```

Why does this happen? Suggest a way (e.g., by defining a new rule) to find a list of people who live near each other, but with each pair appearing only once.

Logic as programs

We can regard a rule as a kind of logical implication: *If* some data satisfies the condition, *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 this chapter. As we said, *append* can be characterized by the following two rules:

1. For any *x*, *y*, *z*, and *t*,
   - if *x* and *y* *append* to form *z*
   - then *(cons t x)* and *y* *append* to form *(cons t z)*
2. For any list *y*, *(append nil y)* is *y*

To express this 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 (?t . ?x) ?y (?t . ?z))
      (append-to-form ?x ?y ?z))

(rule (append-to-form nil ?y ?y))
```

Observe that the first rule makes use of the dot notation introduced above. The second rule has no *condition* part, which means that the conclusion holds for any value of *x*.

Given these two rules, we can formulate queries that compute the *append* of two lists:

```
query-->(append-to-form (a b) (c d) ?z)
(append-to-form (a b) (c d) (a b c d))
```

More strikingly, we can use the same rules to ask the question "which list, when appended to *(a b)*, yields *(a b c d)*"

```
query-->(append-to-form (a b) ?y (a b c d))
(append-to-form (a b) (c d) (a b c d))
```

We can also ask for all pairs of lists that *append* to form *(a b c d)*:

```
query-->(append-to-form ?x ?y (a b c d))
(append-to-form nil (a b c d) (a b c d))
(append-to-form (a) (b c d) (a b c d))
(append-to-form (a b) (c d) (a b c d))
(append-to-form (a b c) (d) (a b c d))
(append-to-form (a b c d) nil (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 unravelling the rules. In addition, 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 4.3.4 below.

Exercise 4-22: Consider the following geneological data base (cf. *Genesis* 4):

```
(wife Adam Eve)
(son Adam Cain)
(son Cain Enoch)
(son Enoch Irad)
(son Irad Mehujael)
(son Mehujael Methushael)
(son Methushael Lamech)
(wife Lamech Ada)
(wife Lamech Zillah)
(son Ada Jabal)
(son Ada Jubal)
(son Zillah Tubal-cain)
(daughter Zillah Naamah)
```

Formulate rules such as "if S is a son of P, then P is a parent of S" or "if P is a parent of S and Q is a parent of P, then Q is a grandparent of S," or "if A is married to B, and A is a parent of S, then B is a parent of S" (more true in biblical times than now) that will enable the query system to deduce information such as

- the grandparents of Enoch
- the sons and daughters of Lamech
- the grandparent of Tubal-cain
- the ancestors of Zillah

4.3.2. How the Query System Works

In section 4.4 below we will present an implementation of the query interpreter as a collection of procedures in Scheme. In this section we give an overview which explains the general structure of the system independently of low-level implementation details.

The query system is built around two central operations called *pattern matching* and

unification. We begin by discussing pattern matching, and 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. The entire query interpreter fits together through a procedure *qeval*, which classifies expressions analogously to the way *eval* classifies expressions for the Lisp interpreter described in the previous chapter. After describing the implementation of the interpreter, we will be 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.

Pattern matching

A *pattern matcher* is a program that tests whether some *datum* fits a specified *pattern*. For example, the data list

```
((a b) c (a b))
```

matches the pattern

```
(?x c ?x)
```

with the pattern variable *x* bound to the list *(a b)*. The same data list matches the pattern

```
(?x ?y ?z)
```

with *x* and *z* both bound to *(a b)* and *y* bound to *c*. It also matches

```
((?x ?y) c (?x ?y))
```

with *x* bound to *a* and *y* bound to *b*. But it does not match the pattern

```
(?x a ?y)
```

since that pattern specifies a list whose second element is the atom *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 to see if 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

```
(?x ?y ?x)
```

to match the datum *(a b a)* 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, whose implementation as a Lisp procedure is given in section 4.4.3, is essentially all the mechanism that is needed to process simple queries. For instance, to process the query

```
(job ?x (computer programmer))
```

we scan through all entries in the data base and select the assertions that match the pattern with respect to an initially empty frame. Then, for each match we find, we use the frame

returned by the match to instantiate the pattern with a value for *x* in the pattern and print the result.

Compound queries and streams of frames

In the query system, testing patterns against frames is organized using 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 else 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.[21]

In our system, a query is represented as a processor on an input *stream of frames*, that performs the above operation for every frame in the stream, as indicated in figure 4-3. 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 of these streams are then appended to form one huge stream, that contains all possible extensions of every frame in the input stream. This stream is the output of the query.

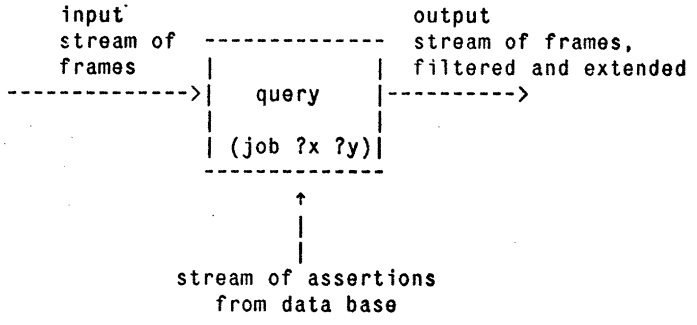


Figure 4-3: Queries are implemented as processors on streams of frames.

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, i.e., 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 at the terminal.

The real elegance of the stream of frames implementation comes when we deal with *compound* queries. This makes use of the ability of our matcher to demand that a match be

---

[21] Because matching is, in general, very expensive, we would like to arrange 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 is used to filter the database to produce a small set of candidates for the final matcher. With care, we can arrange our database so that some of the work of coarse matching can be done at data base construction time, rather then when we want to select the candidates. This is called *indexing* the data base. There is a whole vast technology built around data base indexing schemes. Our implementation will contain a simple-minded form of such an optimization, described below in section 4.4.5.

consistent with a specified frame. For example, to handle the *and* of two queries such as

```
(and (can-do-job ?x (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 (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 patterns 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*. In other words, the *and* of two queries can be viewed as a series combination of the two component queries, as shown in figure 4-4. The frames that pass through the first query filter are then filtered and further extended by the second query.

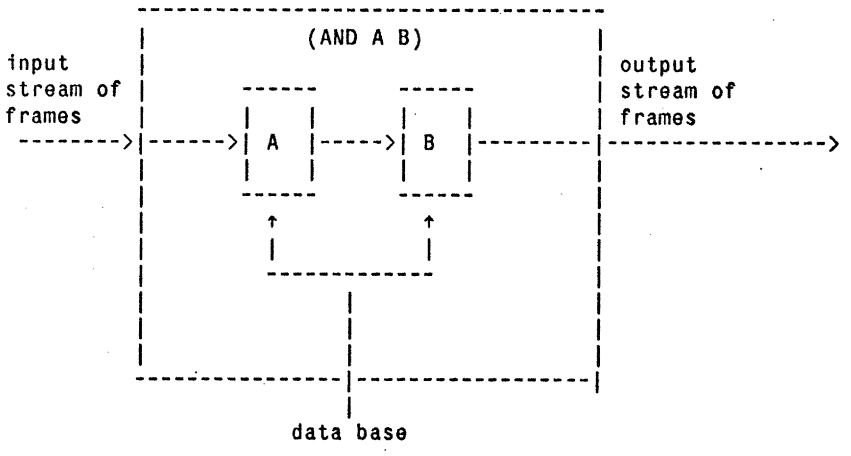


Figure 4-4: The compound query AND of two queries is produced by operating on the frame stream in series.

Figure 4-5 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 (e.g., by appending the streams) to produce the final output stream.

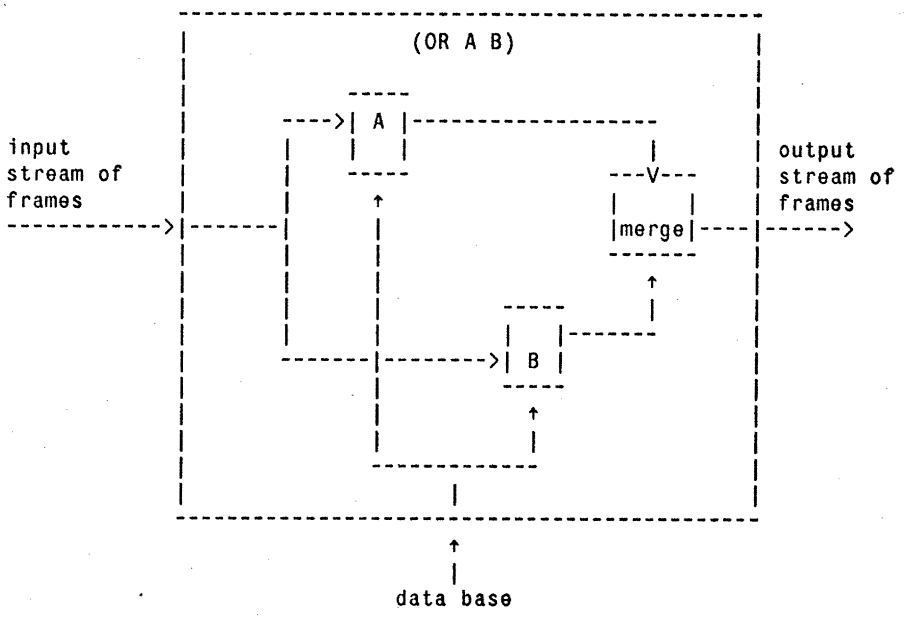


Figure 4-5: The OR combination of two queries is formed by operating on the frame stream in parallel and merging the results.

As vague as this description is, you should already be able to see from it that processing compound queries can be a slow process. In general, if there are D items in the data base, and n clauses in the compound query, we can expect to check on the order of Dn matches. This means that as we form more and more complex queries by compounding elementary queries, the amount of computation required grows considerably. So, while systems for handling only simple queries are quite practical, it is generally agreed that dealing with complex queries will require developing new computer architectures that allow one to apply other strategies, based on parallel processing, to the information retrieval problem.

Unification and applying rules

In order to handle rules in the query language, we must go beyond one-sided pattern matching, to a more general operation called *unification*. This is similar to pattern matching, except that *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 place restrictions on the values of the variables which will make the two patterns be equal. If so, it returns a frame that describes these restrictions. For example,

```
(unify '(?x a ?y) '(?y ?z a))
```

will specify a frame in which x, y and z must all be bound to a. On the other hand,

```
(unify '(?x ?y a) '(?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 a, but for the third elements to be

equal, *y* would have to be *b*.)[22] In addition, the unifier used in the query system, like the pattern matcher, also takes as input an initial frame, and performs unifications that are consistent with this frame.

The unification algorithm, discussed in section 4.4.4 below, is the most technically difficult part of the query system. With complex patterns, performing unification may seem to require deduction. For example,

```
(unify '(?x ?x)) '((a ?y c) (a b ?z)))
```

has to infer that *x* should be *(a b c)*, *y* should be *b* and *z* should be *c*. We may think of matching as setting up a set of equations among the pattern components. In the one-sided match, all of the equations which contain variables are explicit and solved for the unknown (pattern variable). In the 2-sided unify match, the equations are simultaneous and may require substantial manipulation to solve. For example,

```
(unify '(?x ?x)) '((a ?y c) (a b ?z)))
```

may be thought of as a way of writing the following simultaneous equations:

```
?x = (a ?y c)  and  ?x = (a b ?z)
```

We can see that this implies that:

```
(a ?y c) = (a b ?z)
```

Which, in turn, implies that:

```
a = a;  ?y = b;  and  c = ?z
```

Unification is the key to the component of the query system that forms inferences from rules.[23] To see how this is accomplished, consider processing a query that involves applying some rule, for example,

```
(lives-near ?x (Hacker Alyssa P))
```

To process this query, we first use the ordinary pattern match procedure 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 pattern with the conclusion clause of each rule. We find that the pattern unifies with the conclusion clause of the rule

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (lisp-value equal? ?person-1 ?person-2))))
```

resulting in a frame specifying that *person-2* should be bound to *(Hacker Alyssa P)* and

---

[22] 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 *(?x a ?y)* and *(?y ?z a)* is *(a a a)*. For our implementation, it is more convenient to think of the result of unification as being a frame rather than a pattern.

[23] If we built the query system to include only matching, but not unification, then we would be able to process both simple and compound queries, but not rules. Such an information retrieval system is a worthwhile program, though not as interesting in its logical capabilities.

that *x* should 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 interpreter uses the following method to apply a rule in trying to establish a query pattern (in a given 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 *procedure* in *eval/apply* interpreter for Lisp.

- Bind the procedure's parameters to its arguments to form a frame that extends the original procedure frame.
- Relative to the extended frame, evaluate the expression formed by the body of the procedure.

4.3.3. The Query Evaluator

Despite the complexity of the underlying matching operations, the general organization of the system is much like an evaluator for any language. The procedure that coordinates the matching operations is called *qeval*, and it plays a role analogous to that of the *eval* procedure for Lisp.

Qeval takes as inputs a query together with a *stream* of frames. Its output is a stream of frames consisting of successful matches to the query pattern that extend some frame in the input stream, as indicated in figure 4-3.

Like *eval*, *qeval* classifies the different types of expressions and dispatches to an appropriate procedure. For example, *and* queries are handled by a procedure called *conjoin*, Other special forms that are handled through the dispatch are *or*, *not*, *lisp-value*, and *always-true*.[24]

Handling assertions

If no procedure is found under the *qeval* property of the *type* of the expression, the expression is assumed to be an ordinary query, rather than a special form, and *qeval* refers the problem to *asserted?*, which handles ordinary assertions. *Asserted?* takes as input a pattern and a stream of frames. For each frame in the input stream, *asserted?* produces

---

[24] *Always-true* is part of the internal implementation used for rules whose conclusions are specified to be always true (by giving a null body) as in

```
(rule (append-to-form nil ?y ?y))
```

used in the *append* example in section 4-21.

1. a stream of extended frames, which is obtained by matching the pattern and the given frame against all assertions in the data base (using the pattern matcher); and
2. a stream of extended frames obtained by applying all possible rules (using the unifier)[25]

Appending these two streams produces a stream that consists of all ways that the given assertion 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 possible ways that any of the frames in the original input stream can be extended to produce a match with the given pattern. Details of the procedures can be found in section 4.4.

Compound queries

Compound queries are implemented using the series-parallel' combination idea explained in section 4.3.2 and illustrated in figures 4-4 and 4-5. Here, for example is the procedure *conjoin* that handles *and*:

```
(define (conjoin conjuncts frame-stream)
  (cond ((empty-conjunction? conjuncts)
         frame-stream)
        (else (conjoin (rest-conjuncts conjuncts)
                       (qeval (first-conjunct conjuncts)
                              frame-stream)))))
```

The idea is that we first filter the stream of frames by finding the stream of *all* possible extensions to the first clause in the conjunction. Then, using this as the new frame stream, we recursively *conjoin* the rest of the clauses. Since at each step we produce the stream of all possible extensions, we can be sure that we haven't missed any possibilities that might satisfy the conjunction. *Or* expressions are implemented in a similar manner. (See section 4.4 for details.)

Handling NOT

From the stream of frames point of view, the *not* of some query acts as a filter, which removes all frames for which the query can be satisfied. For instance, given some frame stream, and the pattern

```
(not (job ?x (computer programmer)))
```

we attempt, for each frame in the stream, to produce extension frames that satisfy *(job ?x (computer programmer))*. We remove from the input stream all frames for which such extensions exist. The result is a stream of consisting of only those frames in which the binding for *x* does not satisfy *(job ?x (computer programmer))*. Thus, for example, in processing the query

---

[25] Since unification is a generalization of matching, we could simplify the system by using the unifier to produce both streams. On the other hand, the full-blown unification algorithm requires much more work than the simple matcher, and our system will run more efficiently if we use simple matching wherever we can get away with it.

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))
```

the first form will generate frames with bindings for *x* and *y*. Taking *and* with the *not* form will filter these by removing all frames in which the binding for *x* satisfies the restriction that *x* is a computer programmer.[26] The *lisp-value* special form is implemented as similar filter on frame streams. (See section 4.4.)

The driver loop

The *query-driver-loop* procedure, analogous to the *driver-loop* procedure of the interpreter in chapter 4, reads queries from the terminal. For each query, it calls *qeval* with the query and a frame stream that consists of a single empty frame. This will produce the stream of all possible matches (i.e., all possible extensions to the empty frame). For each frame in the resulting stream, we instantiate the original query using the values of the variables found in the frame. This stream of instantiated queries is then printed, one item to a line. In our implementation, the driver also performs some minor syntactic processing to mediate between the form of the expression that the user types and an internal form that is more convenient for the implementation. In addition, the driver 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:

```
query-->(assert! (job (Bitdiddle Ben) (computer wizard)))

query-->(assert!
         (rule (wheel ?person)
               (and (supervisor ?middle-manager ?person)
                    (supervisor ?x ?middle-manager))))
```

There is also a special *initialize-data-base* command that loads the data base with a given collection of assertions and rules.

4.3.4. Is Logic Programming Mathematical Logic?

On the face of it, the means of combination of our query language seem identical to the operations *and*, *or*, and *not* or mathematical logic, and the application of query language *rules* is done with a legitimate rule of inference.[27] However, this is not quite true, because our 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 the two logically equivalent forms:

---

[26] There is a subtle difference between this filter implementation of *not* and the usual meaning of *not* in mathematical logic. See section 4.3.4 below.

[27] The statement that a rule is a legitimate rule of inference, is not trivial. It is required to prove that, starting with true premises, only true conclusions may be derived. The particular rule of inference we are using here is a generalized form of *modus ponens*, called *cut*. *Modus ponens* is the familiar rule which says that if *A* is true and if *A* implies *B* is true, then we may conclude that *B* is true.

```
(and (job ?x (computer programmer))
     (supervisor ?x ?y))

(and (supervisor ?x ?y)
     (job ?x (computer programmer)))
```

If the a company has many more supervisors than programmers[28] it would be advantageous to use the first form rather than the second one. This is because the data base must be scanned for each intermediate result produced by the first clause of the *and*.

In fact, it is the aim of Logic Programming is to provide the programmer with techniques for decomposing a computational problem into two separate problems: *what* he is trying to compute and *how* he wants to go about it. The idea is to select a subset of the statements of mathematical logic which are just powerful enough to be able to describe anything one might want to compute, but which are weak enough to have a controllable procedural interpretation. Implications with only one consequent (called *Horn clauses*) meet both requirements. Because the Horn clauses can be interpreted as statements of mathematical logic, we can verify, using standard proof techniques, that a particular assertion means what we intend it to and that establishing its truth in fact computes what we wanted. Thus, *if* a logic program terminates, we can be sure of the result.[29] Control is effected by using the order of evaluation of the language. We arrange the order of clauses, and the order of subgoals within each clause, so that the computation is done in an order which we deem to be effective and efficient.

Infinite loops

A consequence of the ability to control the execution of a logic program by imposing order on the clauses, is that it is possible to construct hopelessly inefficient programs for solving certain problems. An extreme case of inefficiency is occurs when the system falls into infinite loops in making deductions. As a simple example, suppose we are setting up a data base about famous marriages, including

```
query-->(assert! (married Minnie Mickey))
```

If we now ask

```
query-->(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 following rule:

```
query-->(assert! (rule (married ?x ?y)
                       (married ?y ?x)))
```

and query again

```
query-->(married Mickey ?who)
```

Unfortunately, this will drive the system into an infinite loop, as follows:

---

[28] the usual case

[29] Even this statement is false in our query language (and also false for programs in Prolog) because of our use of `lisp-value` and `not`, as described below.

1. The system finds that the *married* rule is applicable, i.e., the rule conclusion -- *(married ?x ?y)* -- successfully unifies with the query to produce a frame in which *x* is bound to *Mickey*. So the interpreter proceeds to evaluate the rule body -- *(married ?y ?x)* -- in a frame in which *x* is bound to *Mickey*, in effect, to process the query *(married ?y Mickey)*.
2. One answer is found directly as an assertion in the data base -- *(married Minnie Mickey)*.
3. But the *married rule* is also applicable, because the rule conclusion -- *(married ?x ?y)* -- unfies with the query -- *(married ?y ?x)* in the frame -- *x* bound to *Mickey* -- to produce a frame in which *x* and *y* are both bound to *Mickey*. Now, in this new frame, the system proceeds to process the rule body -- *(married ?y ?x)*, in effect to evaluate the query *(married Mickey Mickey)*.
4. Once again, the *married* rule, which leads the interpreter to again evaluate the rules body, and so on.

The system is in an infinite loop. Indeed, whether the system will find the simple answer in step 2 above before it goes into the loop depends on implementation details concerning the order in which it checks the items in the data base. This is a very simple example of the kinds of loops that can occur. Collections of inter-related rules can lead to loops that are much harder to anticipate, and the appearance of a loop can depend on low-level details concerning the order in which the system processes queries.[30]

Exercise 4-23: While Louis Reasoner is using the personnel data base, he mistakenly deletes the *outranked-by* rule, given above in section 4-18. When he realizes this, he quickly re-installs it. Unfortunately, he makes a slight change in the rule, typing 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, Rosemary Forrest comes by to find out who is outranked by Ben Bitdiddle, and issues the query

```
(outranked-by (Bitdiddle Ben) ?who)
```

Instead of answering, the system goes into an infinite loop. Explain why.

Exercise 4-24: Cy D. Fect, looking forward to the day when he will rise in the organization, adds to the personnel data base the rule for determining who is a wheel:

---

[30] Note that this is not a problem of the *logic*, but rather of the *procedural interpretation of the logic* provided by our interpreter. One could write an interpreter that would not fall into a loop here. For example, we could enumerate all of 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 [10]. 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 4-26). However, that there can in general be no scheme for reliably preventing a system from tracing down infinite paths in performing deductions. Image 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*.

```
(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
           (supervisor ?x ?middle-manager)))
```

To test the rule, he gives a query to find all people who are wheels:

```
query-->(wheel ?who)
```

To his surprise, the system responds:

```
(wheel (Warbucks Oliver))
(wheel (Bitdiddle Ben))
(wheel (Warbucks Oliver))
(wheel (Warbucks Oliver))
(wheel (Warbucks Oliver))
```

Why is Oliver Warbucks listed four times?

Exercise 4-25: Ben has been generalizing the query system to provide all kinds of statistics about the company. For example, to find the total salaries of all the computer programmers, one will be able to type:

```
(sum ?amount
     (and (job ?x (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, maximum, and so on. 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 4-24. 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 that he can use to salvage the situation.

Exercise 4-26: Devise a way to install a loop detector in the query system so as to avoid the kinds of simple loops illustrated above in the text and in exercise 4-23. The general idea is that the system should maintain some sort of history of its current chain of deductions, and 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, you may want to actually modify the system to include your loop detector.)

Problems with NOT

Another quirk in the query system concerns *not*. Given the data base of section 4.3.1 Consider the two queries:

```
(and (supervisor ?x ?y)
     (not (job ?x (computer programmer))))

(and (not (job ?x (computer programmer)))
     (supervisor ?x ?y))
```

The 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 filtering these by removing the ones in which the value of *x* satisfies *(job ?x (computer programmer))*.

But the second query begins by checking the data base to see if there are any patterns that satisfy *(job ?x (computer programmer))*. Since there are, in general, entries of this form, the *not* clause returns an empty stream of frames, and consequently the entire compound query returns an empty stream.

The problem 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 using 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 *lisp-value*. See exercise 4-29.

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 4.3.1, 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, that two plus two is not four, and so on. 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.

4.4. Implementing the Query System

Section 4.3.2 described, in outline, how the query system works. Now we fill in the details, by presenting a complete implementation of the query system as a collection of procedures in Scheme.

4.4.1. Driver Loop and Syntax Processing

The driver loop for the query system reads expressions from the terminal. If the expression indicates that this 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 pattern with variable values found in the data base. These frames are used to form a new stream, consisting of copies of the original query, where the variables are instantiated with values supplied by a stream of frames, and this final stream is printed at the terminal.

```
(define (query-driver-loop)
  (newline)
  (let ((q (query-syntax-process (read 'query-->))))
    (if (assertion-to-be-added? q)
        (sequence (add-assertion! (add-assertion-body q))
                  (print "assertion added to data base")
                  (query-driver-loop))
        (sequence
         (print-stream-elements-on-separate-lines
          (map (lambda (frame) (instantiate q frame))
               (qeval q (singleton '()))))
         (query-driver-loop)))))
```

In addition, before doing any processing on an input expression, the driver loop makes a syntax transformation, to transform pattern variables of the form *?symbol* into the internal

format *(? symbol)*. That is to say, a pattern such as

```
(job ?x ?y)
```

is actually represented internally by the system as

```
(job (? x) (? y))
```

This increases the efficiency of the internal processing, since it means that the system can check to see if an expression is a pattern variable by checking whether the *car* of the expression is the symbol *?*, rather than having to extract characters from the symbol, which is much less efficient. The syntax transformation is accomplished by the following procedure:[31]

```
(define (query-syntax-process exp)
  (map-over-atoms expand-question-mark exp))

(define (expand-question-mark symbol)
  (let ((characters (explode symbol)))
    (if (eq? (car characters) '?)
        (list '? (implode (cdr characters)))
        symbol)))

(define (map-over-atoms proc exp)
  (cond ((null? exp) nil)
        ((pair? exp) (cons (map-over-atoms proc (car exp))
                           (map-over-atoms proc (cdr exp))))
        ((atom? exp) (proc exp))
        (else (error "unknown expression -- Map over atoms"
                     exp))))
```

(The program uses the primitives *explode*, which separate a symbol into a list of characters, and *implode*, which assembles a list of single characters to form a new symbol. Notice how we have abstracted out the control structure *map-over-atoms*, which applies a procedure to every atomic symbol in a list.)

4.4.2. The Evaluator

The *qeval* procedure, already described in section 4.3.3, is the the basic evaluator of the query system. As described above, it takes as inputs a query and a stream of frames, and 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 not identified as a special form is assumed to be a simple assertion to be checked.

```
(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if (not (null? qproc))
        (qproc (contents query) frame-stream)
        (asserted? (make-arg-list query)
                   frame-stream))))
```

---

[31] Most Lisp systems, in fact, give the user the ability to modify the ordinary *read* procedure to perform such transformations by defining *reader macro characters*. Quoted expressions are already handled in this way, since the reader automatically translates *'expression* into *(quote expression)* before the evaluator does any further processing. We could arrange for *:expression* to be transformed into *(? expression)* in just the same way, but, for the sake of clarity, we have included the transformation procedure here explicitly.

Simple queries

The *asserted?* procedure handles simple queries. It takes an argument list containing one assertion, together with a stream of frame, and returns the stream formed by extending each frame by a data-base match of the query. To accomplish this, it uses the procedure *find-assertions*, which generates, for each frame, a stream of extended frames. To apply *find-assertions* to each frame in the input stream, and consolidate the output frames for each frame into one large output stream, we apply the *flatmap* procedure, which was introduced in Chapter 3, section 3.4.2 to perform just this kind of mapping and accumulation. Similarly, a procedure *apply-rules* is used to generate a stream of extensions *F*, found by applying rules, for each frame in the input stream, and the results are accumulated using *flatmap*. Finally, the two streams, one generated by checking the assertions and one generated by applying the rules, are appended to form a single output stream.

```
(define (asserted? a frame-stream)
  (append-streams
   (flatmap (lambda (frame)
              (find-assertions (pattern-of a) frame))
            frame-stream)
   (flatmap (lambda (frame)
              (apply-rules (pattern-of a) frame))
            frame-stream)))
```

Compound queries

And queries are handled by a *conjoin* operation, as described above. This takes as inputs the list of conjunctions and the frame stream, and returns the stream of extended frames.

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts)
                      frame-stream))))
```

The following *put* expression sets up *qeval* to dispatch to *conjoin* when an *and* form is encountered:

```
(put 'and 'qeval conjoin)
```

Or forms are handled similarly, according to the diagram in figure 4-5. The output streams for the various disjuncts of the *or* are computed separately and then merged, using *append-streams*:

```
(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
       (append-streams (qeval (first-disjunct disjuncts)
                              frame-stream)
                       (disjoin (rest-disjuncts disjuncts)
                                frame-stream))))

(put 'or 'qeval disjoin)
```

Filters

Negations are handled by the method outlined in section 4.3.3. We attempt to extend each frame in the input stream, and we include a given frame in the output stream only if it cannot be extended.

```
(define (negate a frame-stream)
  (flatmap
   (lambda (frame)
     (if (empty-stream? (qeval (expression-argument a)
                               (singleton frame)))
         (singleton frame)
         the-empty-stream))
   frame-stream))

(put 'not 'qeval negate)
```

Lisp-value is a filter similar to *not*. For each frame in the stream is used to instantiate the variables in the pattern, the indicated predicate is applied, and the frame for which the predicate returns *nil* are filtered out of the input stream:

```
(define (lisp-value call frame-stream)
  (flatmap
   (lambda (frame)
     (let ((lcall (instantiate call frame)))
       (if (execute lcall)
           (singleton frame)
           the-empty-stream)))
   frame-stream))

(put 'lisp-value 'qeval lisp-value)
```

The *always-true* special form (which signals that the associated clause is always true simply ignores the pattern, and passes through all the frames in the input stream:

```
(define (always-true ignore frame-stream)
  frame-stream)

(put 'always-true 'qeval always-true)
```

4.4.3. Pattern Matching and Finding Assertions

Here is the basic pattern matcher. It takes as arguments a pattern, a data object, and a frame and returns either a stream containing the extended frame, or *nil* if the match fails:

```
(define (pattern-match pat dat frame)
  (let ((result (internal-match pat dat frame)))
    (if (eq? result 'failed)
        the-empty-stream
        (singleton result))))
```

The main *pattern-match* procedure calls *internal-match*, which returns either the symbol *failed*, or an extension of the given frame:

```
(define (internal-match pat dat frame)
  (cond ((eq? frame 'failed) 'failed)
        ((and (number? pat) (number? dat))
         (cond ((= pat dat) frame)
               (else 'failed)))
        ((atom? pat)
         (cond ((eq? pat dat) frame)
               (else 'failed)))
        ((var? pat)
         (extend-if-consistent pat
                               dat
                               frame))
        ((atom? dat) 'failed)
        (else (internal-match (cdr pat)
                              (cdr dat)
                              (internal-match (car pat)
                                              (car dat)
                                              frame)))))
```

The basic idea of the matcher is to check the pattern against the data, symbol by symbol, accumulating bindings for the pattern variables. We first consider the case where the pattern is an atomic symbol. If so, then either it is equal to the data -- in which case the match succeeds and we return the frame of bindings accumulated so far -- or it is not equal to the data -- in which case the match fails and we return *failed*. The next case is where the pattern is a variable. If so, 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. The next case is where the pattern is not atomic, but the data is, in which case the match must fail. Finally (recursively) if the pattern and the data are both non-atomic, we match the *car* of the pattern against the *car* of the *data* to produce a frame. In this frame, we then match the *cdr* of the pattern against the *cdr* of the data.

Here is the auxiliary procedure that extends a frame by adding a new binding binding, if this is consistent with the bindings already in the frame:

```
(define (extend-if-consistent var dat frame)
  (let ((value-cell (binding-in-frame var frame)))
    (if (null? value-cell)
        (extend var dat frame)
        (internal-match (binding-value value-cell) dat frame))))
```

The check for consistency is rather subtle. 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 and this will return either the original frame, or else a failure indication. The reason for the subsidiary match is to handle the following kind of situation. Suppose we have a frame that specifies that *x* is bound to *y* and *y* is bound to 5, and we wish to augment this frame by a binding of *x* to 5. We look up *x* and find that it is bound to *y*. This leads us to check for consistency by matching the value of *x*, that is, *y*, against the proposed new binding, that is, 5, in the current frame.

The matcher is used by the procedure *find-assertions*, which 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 a subsidiary procedure *fetch-assertions* which returns a stream of all the items 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 possible candidates for a successful match. The system would still work if we eliminated

fetch-assertions, and simply returned a stream of all assertions in the data base, but the computation would be less efficient, since we would need to make many more calls to the matcher.

```
(define (find-assertions pattern frame)
  (flatmap (lambda (datum)
             (pattern-match pattern datum frame))
           (fetch-assertions pattern frame)))
```

4.4.4. Rules and Unification

Apply-rules is the rule analogue of *find-assertions*. It takes as input a pattern and a frame, and forms the stream of extension frames formed by applying rules from the data base. Each applicable rule may return a stream of frames, formed by a procedure *apply-a-rule*. This is mapped down the stream of rules (selected by a procedure *fetch-rules*, analogous to *fetch-assertions*) and the results are accumulated by *flatmap*:

```
(define (apply-rules pattern frame)
  (flatmap (lambda (rule)
             (apply-a-rule rule pattern frame))
           (fetch-rules pattern frame)))
```

Apply-a-rule applies rules using the method outlined in section 4.3.2. It first forms a *rule-frame* obtained by unifying the rule conclusion with the pattern in the given frame. Then it evaluates the rule condition in this new frame. (Actually, since *qeval* is set up to accept a stream of frames, it passes to *qeval* a stream consisting of this single frame.)

Before any of this happens, however, the program first renames all the variables in the rule with unique new names. The reason for this is to prevent the annoying bug that, the frame structures we are accumulating, typically the result of applying many rules, the variables for different rule applications may become 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. But, in fact, these two *x*'s having nothing to do with each other, and we should not be fooled into thinking that the two bindings must be consistent. Rather than renaming variables, we could devise a more clever environment structure, but the renaming approach we have chosen here is the most straightforward, even if not the most efficient. (See exercise 4-30.) Here is the resulting *apply-a-rule* procedure.

```
(define (apply-a-rule rule query-pattern query-frame)
  (let ((clean-rule (rename-variables-in rule)))
    (let ((unify-result (unify-match query-pattern
                                     (rule-conclusion clean-rule)
                                     query-frame)))
      (if (empty-stream? unify-result)
          the-empty-stream
          (qeval (rule-condition clean-rule)
                 unify-result)))))
```

Generating unique variable names is accomplished by maintaining a rule counter, that is incremented at each rule application. This counter is then included as part of the original variable name:

```
(define (rename-variables-in rule)
  (define (tree-walk exp)
    (cond ((atom? exp) exp)
          ((var? exp) (make-new-variable exp))
          (else (cons (tree-walk (car exp))
                      (tree-walk (cdr exp))))))
  (increment-rule-counter)
  (tree-walk rule))

(define rule-counter 0)

(define (increment-rule-counter)
  (set! rule-counter (1+ rule-counter)))

(define (make-new-variable var)
  (cons '? (cons rule-counter (cdr var))))
```

Finally, here is the unification algorithm, implemented as a procedure that takes as inputs two patterns and a frame, and returns either a stream containing an extended frame, or else the empty stream. The actual unification is performed by *internal-unify*, which returns either the extended frame itself, or the symbol *failed*:

```
(define (unify-match p1 p2 frame)
  (let ((result (internal-unify p1 p2 frame)))
    (if (eq? result 'failed)
        the-empty-stream
        (singleton result))))
```

The unifier is just like the pattern matcher except that it is symmetrical -- variables are allowed on both sides of the match. The basic program is exactly the same, except that there are two extra lines (marked "***" below) which test for the possibility that the object on the right side of the match is a variable.

```
(define (internal-unify p1 p2 frame)
  (cond ((eq? frame 'failed) 'failed)
        ((equal? p1 p2) frame)
        ((atom? p1)
         (cond ((atom? p2) 'failed)
               ((var? p2) (extend-if-possible p2 p1 frame))     ;***
               (else 'failed)))
        ((var? p1) (extend-if-possible p1 p2 frame))
        ((atom? p2) 'failed)
        ((var? p2) (extend-if-possible p2 p1 frame))          ;***
        (else (internal-unify (cdr p1)
                              (cdr p2)
                              (internal-unify (car p1)
                                              (car p2)
                                              frame)))))
```

In unification, as in one-sided pattern matching, we only want to accept an extension of the frame if the proposed extension is consistent. For the unifier, we have to consider a few cases that cannot occur in a one-sided matcher. The interesting case arises when the same variable occurs on both sides of a match. This can occur whenever a variable is repeated in both patterns. Consider, for example, the following two patterns:

```
p1 = (?x ?x)   and   p2 = (?y (f ?y))
```

In this case, first *?x* is matched against *?y*, making a binding of *?x* to *?y*. Next, the same *?x* is matched against *(f ?y)*. Since *?x* is already *?y* this is the same as matching *?y* against *(f ?y)*. To put it another way, if we think of the unifier as finding a set of values for the pattern

variables which make the patterns the same, then these patterns imply instructions to find a *?y* such that *?y = (f ?y)* -- a fixed point of *f*. If we could routinely solve such equations in the matcher we would develop the solution, but this is a very difficult technical problem. Thus we reject such a pattern with the predicate *freefor?* (see the line marked "*###*" below).

On the other hand, we do not want to rule out important special cases such as the following renaming of the variable *?x* to *?y*.

```
p1 = (?x ?x)  and  p2 = (?y ?y)
```

In our unifier, we explicitly admit this case by a special check at the beginning of the frame extender (in the line marked "*$$$*" below).

```
(define (extend-if-possible var val frame)
  (if (equal? var val)                          ;$$$
      frame
      (let ((value-cell (binding-in-frame var frame)))
        (if (null? value-cell)
            (if (freefor? var val frame)        ;###
                (extend var val frame)
                'failed)
            (internal-unify (binding-value value-cell)
                            val
                            frame)))))
```

The *freefor?* test is a simple predicate tests to see if an expression proposed to be the value of a pattern variable contains 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 which might contain our test variable. The structure of *freefor?* is a simple recursive tree walk, substituting for the values of variables, whenever necessary.

```
(define (freefor? var exp frame)
  (define (freewalk e)
    (cond ((atom? e) t)
          ((var? e)
           (if (equal? var e)
               nil
               (freewalk (lookup-in-frame e frame))))
          ((freewalk (car e)) (freewalk (cdr e)))
          (else nil)))
  (freewalk exp))
```

4.4.5. Maintaining the Data Base

One important problem in designing logic programming languages is that of arranging things so as few as possible irrelevant data bases entries will be examined in checking a given pattern. In the present system we place all data base entries whose *car*'s are atomic symbols in separate streams, indexed by the symbol. To fetch an assertion that may match an item, we first check to see if the *car* of the item is atomic. If so, we return (to be tested using the matcher) all stored patterns that have the same *car*, and also all stored patterns that have a non-atomic *car*. If the pattern is non-atomic we (uncleverly) return all stored assertions. More clever methods could also take advantage of information in the frame, or try also to optimize the case where the *car* of the pattern is not atomic.

```
(define THE-ASSERTIONS the-empty-stream)
(define ASSERTIONS-WITH-NON-ATOMIC-CARS the-empty-stream)
```

```
(define (fetch-assertions pattern frame)
  (if (atom? (car pattern))
      (get-assertions-on-list (car pattern))
      (get-all-assertions)))

(define (get-all-assertions) THE-ASSERTIONS)

(define (get-assertions-on-list symbol)
  (append-streams
   ASSERTIONS-WITH-NON-ATOMIC-CARS
   (let ((assertion-stream (get symbol 'assertion-stream)))
     (if (null? assertion-stream)
         the-empty-stream
         assertion-stream))))
```

Rules are stored similarly, using the *car* of the rule conclusion:

```
(define THE-RULES the-empty-stream)
(define RULES-WITH-NON-ATOMIC-CARS the-empty-stream)

(define (fetch-rules pattern frame)
  (if (atom? (car pattern))
      (get-rules-on-list (car pattern))
      (get-all-rules)))

(define (get-all-rules) THE-RULES)

(define (get-rules-on-list symbol)
  (append-streams
   RULES-WITH-NON-ATOMIC-CARS
   (let ((rule-stream (get symbol 'rule-stream)))
     (if (null? rule-stream)
         the-empty-stream
         rule-stream))))
```

The data base is initialized from a big list of assertions, as follows:

```
(define (initialize-data-base big-list)
  (define (deal-out statements rules assertions)
    (if (null? statements)
        (sequence (set! THE-ASSERTIONS assertions)
                  (set! THE-RULES rules)
                  'done)
        (let ((s (query-syntax-process (car statements))))
          (if (rule? s)
              (sequence (store-rule-according-to-car s)
                        (deal-out (cdr statements)
                                  (cons s rules)
                                  assertions))
              (sequence
               (store-assertion-according-to-car s)
               (deal-out (cdr statements)
                         rules
                         (cons s assertions)))))))
  (deal-out big-list '() '()))
```

The following procedures are used to add simple assertions and rules:

```
(define (add-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-simple-assertion! assertion)))

(define (add-simple-assertion! assertion)
  (store-assertion-according-to-car assertion)
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS (cons-stream assertion old-assertions))
    'ok))

(define (add-rule! rule)
  (cond
   ((rule? rule)
    (store-rule-according-to-car rule)
    (let ((old-rules the-rules))
      (set! THE-RULES (cons-stream rule old-rules))
      'ok))
   (else (error "badly formed rule"))))
```

To actually store a simple assertion, we check to see if its *car* is an atom. If so, we store it on the appropriate list and also in the list of all simple assertions. Simple assertions with non-atomic *car*s are kept on a special list.

```
(define (store-assertion-according-to-car assertion)
  (cond ((not (atom? (car assertion)))
         (let ((old-assertions ASSERTIONS-WITH-NON-ATOMIC-CARS))
           (set! ASSERTIONS-WITH-NON-ATOMIC-CARS
                 (cons-stream assertion old-assertions))))
        (else
         (let ((current-assertion-stream
                (get (car assertion) 'assertion-stream)))
           (cond ((null? current-assertion-stream)
                  (put (car assertion)
                       'assertion-stream
                       (singleton assertion)))
                 (else
                  (put (car assertion)
                       'assertion-stream
                       (cons-stream assertion
                                    current-assertion-stream))))))))
```

Rules are stored similarly, under the *car* of the rule pattern:

```
(define (store-rule-according-to-car rule)
  (let ((pattern (rule-conclusion rule)))
    (cond ((not (atom? (car pattern)))
           (let ((old-rules RULES-WITH-NON-ATOMIC-CARS))
             (set! RULES-WITH-NON-ATOMIC-CARS
                   (cons-stream rule old-rules))))
          (else
           (let ((current-rule-stream
                  (get (car pattern) 'rule-stream)))
             (cond ((null? current-rule-stream)
                    (put (car pattern)
                         'rule-stream
                         (cons-stream rule
                                      the-empty-stream)))
                   (else
                    (put (car pattern)
                         'rule-stream
                         (cons-stream rule
                                      current-rule-stream)))))))))
```

4.4.6. Utility Procedures

Finally, we need some utility procedures for performing low-level operations, implementing streams, and so on.

The first procedure copies an expression, replacing any variables in the expression by their values in a given frame:

```
(define (instantiate exp frame)
  (define (copy exp)
    (cond ((atom? exp) exp)
          ((var? exp)
           (let ((vcell (binding-in-frame exp frame)))
             (cond ((not (null? vcell))
                    (copy (binding-value vcell)))
                   (else exp))))
          (else (cons (copy (car exp))
                      (copy (cdr exp))))))
  (copy exp))
```

The next procedure, used in the implementation of *`lisp-eval`*, evaluates an expression who *`car`* is interpreted as the name of a Lisp procedure.

```
(define (execute exp)
  (apply (eval (car exp) (the-environment))
         (cdr exp)))
```

Stream operations

We make use of the following stream operations:

```
(define (interleave s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (head s1)
                   (interleave s2
                               (tail s1)))))
```

```
(define (flatmap f s)
  (if (empty-stream? s)
      the-empty-stream
      (let ((s1 (f (head s))))
        (if (empty-stream? s1)
            (flatmap f (tail s))
            (cons-stream (head s1)
                         (interleave (flatmap f (tail s))
                                     (tail s1)))))))

(define (singleton s) (cons-stream s the-empty-stream))

(define (map proc s)
  (if (empty-stream? s)
      the-empty-stream
      (cons-stream (proc (head s))
                   (map proc (tail s)))))

(define (append-streams s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (head s1)
                   (append-streams (tail s1) s2))))
```

Frame access operations

Frames are represented as lists of pairs, just as in our Lisp evaluator of section 4.1.3.

```
(define (make-binding variable value)
  (cons variable value))

(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (binding-in-frame variable frame)
  (assoc variable frame))

(define (extend variable datum frame)
  (cons (make-binding variable datum) frame))

(define (unbound? var frame)
  (null? (binding-in-frame var frame)))

(define (lookup-in-frame variable frame)
  (binding-value (binding-in-frame variable frame)))
```

Syntax procedures

```
(define (type exp)
  (cond ((not (atom? exp))
         (cond ((atom? (car exp)) (car exp))
               (else nil)))
        (else (error "unknown expression type" exp))))

(define (contents exp)
  (cond ((not (atom? exp)) (cdr exp))
        (else (error "unknown expression contents" exp))))
```

```
(define (assertion-to-be-added? exp)
  (eq? (type exp) 'assert!))

(define (add-assertion-body exp) (cadr exp))

(define pattern-of car)
(define expression-argument car)
(define (make-arg-list arg) (list arg))

(define empty-conjunction? null?)
(define first-conjunct car)
(define rest-conjuncts cdr)

(define empty-disjunction? null?)
(define first-disjunct car)
(define rest-disjuncts cdr)

(define (rule? statement)
  (and (not (atom? statement))
       (eq? (car statement) 'rule)))

(define rule-conclusion cadr)

(define (rule-condition rule)
  (if (null? (cddr rule))
      '(always-true)
      (caddr rule)))

(define (var? exp) (eq? (car exp) '?))
```

Printing utility

```
(define (print-stream-elements-on-separate-lines s)
  (if (empty-stream? s)
      (print "done")
      (sequence (print (head s))
                (print-stream-elements-on-separate-lines
                 (tail s)))))
```

Exercise 4.27: Implement for the query language a new special form called *unique*. The idea of *unique* is that it succeeds if there is *precisely one* item in the data base satisfying the specified condition. For example,

```
query-->(unique (job ?x (computer wizard)))
```

should print the stream of one item

```
(unique (job (Bitdiddle Ben) (computer wizard)))
```

since Ben is the only computer wizard, while

```
query-->(unique (job ?x (computer programmer)))
```

should print the empty stream, since there is more than one computer programmer. Moreover, a query such as

```
query-->(and (job ?x ?j)
             (unique (job ?anyone ?j)))
```

should list all the jobs that are filled by only one person, together with the people that fill them.

There are two parts to implementing *unique*. The first is to write a procedure that handles this form, and the second is to make *qeval* dispatch to your procedure. The second part is trivial, since *qeval* does its dispatching in a data-directed way. Assuming that your procedure is called, say

uniquely-asserted?, all you need do is include

```
(put 'unique 'qeval uniquely-asserted?)
```

and *qeval* will now dispatch to this procedure for every query whose type (*car*) is the symbol *unique*.

The real problem is to write the procedure *uniquely-asserted?*. This should take as input the *cdr* of the *unique* query, together with a *stream of frames*. What it should do is extract the pattern from the query handed in, and then, for each frame in the stream, it should use *qeval* find the stream of all extensions to the frame produced by matching the pattern in the data base. Any stream that does not have exactly one item in it should be eliminated. For each remaining stream, its single frame should be passed back to be accumulated into one big stream that is the result of the query. This is similar to the implementation of the *not* special form.

Test your implementation by forming a query that finds all people who supervise precisely one person.

Exercise 4-28: Our implementation of *and* as parallel combination of queries (section 4.3.2) is elegant, but inefficient. This is because, in processing the second clause of the *and* we must scan the data base for *each* frame produced by the first query. For example, if the data base has N elements, and a typical query produces a number of output frame proportional 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. In contrast, an alternative structure where we process the two clauses of the *and* separately, and 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 procedure that takes two frames as inputs, checks whether the bindings in the frame are compatible, and, if so, produces a frame that merges the two sets of bindings. Notice that this operation is similar to unification.

Exercise 4-29: In section 4.3.4 we saw that *not* and *lisp-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 problem. One idea is to perform the filtering in a "delayed" manner, by appending to the frame a "promise" to filter that is only fulfilled when enough variables have been bound to make the operation possible. One could wait to perform filtering until all other operations have been performed. However, for efficiency's sake, one would like to perform filtering as soon as possible so as to cut down on the number of intermediate frames generated.

Exercise 4-30: When we implemented the Lisp evaluator in section 4.1, we saw how to use local environments to avoid name conflicts between the parameters of procedures. For example, in evaluating

```
(define (square x)
  (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square x)))

==>(sum-of-squares 3 4)
```

there will be no confusion between the x in *square* and the x in *sum-of-squares*, because we evaluate the body of each procedure 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. As explained in section 4.4.4, each time we apply a rule, we rename the variables new names that are guaranteed to be unique. The analogous strategy for the Lisp evaluator would be to do away with local environments and simply rename the variables in the body of a procedure each time we apply the procedure. On the other hand, we saw that the use of environments can lead to important tools for structuring programs, because environments furnish a *context* in which computations can take place. One example of this is block structure. Another is the packaging mechanism discussed in appendix I.

Implement for the query language a rule-application method that uses environments rather than substitution. Now see if you can build on your environment structure to create constructs in the query

language for dealing with large systems, such a the rule analog of block-structured procedures. 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 very open-ended. A good answer is probably worth a Ph.D. thesis.)

Chapter 5
Register Machine Model of Control

Our examination of the meta-circular interpreter has stripped away much of the magic from how a Lisp-like language is interpreted. But the meta-circular evaluator fails to elucidate the mechanisms of control in a Lisp system. For example, it does not explain how the evaluation of a subexpression manages to return its value to the expression which is waiting for that value. This is because the meta-circular evaluator inherits the control structure of the underlying Lisp system. In section 5.2 we will fill in that gap by providing an explicit model for the mechanisms of control in a Lisp evaluator.

Specifically, we will develop an evaluator that is implemented in a style that matches the step-by-step operation of a traditional computer. Such a computer, or *register machine*, sequentially executes *instructions* that manipulate the contents of a small fixed set of *registers*. A typical instruction might assign to one register the result of applying a primitive operation to the contents of one or two other registers. Other instructions (called "branches") conditionally continue at one of two instruction streams depending upon the value of a primitive predicate.

Some of the "primitive operations" are quite simple, such as incrementing the value of the number stored in a register. Such an operation can be performed in easily described hardware. Other operations we will assume here are not so simple. For example, *car*, *cdr*, and *cons* are complex "memory" operations, which are primitive from the point of view of the evaluator, but which may require an elaborate list structure storage mechanism to back them up.[1] We start by assuming them to be primitive. Later, in section 5.3 we will study their implementation in terms of more primitive operations.

Strategically, we will begin this chapter as a hardware architect rather than as a machine-language user of a computer. We will not learn any particular computer machine language. Instead, we will examine several algorithms we wish to compute, expressed in SCHEME, and we will *design* simple computers specifically to execute those algorithms. In doing so we will develop structures and conventional organizations for implementing particular programming constructs, such as recursion. Our knowledge will have the flavor of a parts catalog or data book rather than that of a user's reference manual.

We will also develop a register-transfer machine language for expressing our designs. This will consist of two parts, the description of the data paths (registers and operators) of our machine, and the description of the controller, which sequences those operations. When we have accumulated enough experience, we will design a machine which directly executes the algorithm described by our meta-circular interpreter. Such a machine is a *universal engine* in that it can simulate any other machine whose behavior can be expressed in SCHEME code.

Later in the chapter, once we have an interpreter, we will take our interpreter machine's data paths to be fixed. The controller description part of our machine language will then

---

[1] The management of list structure in a computer memory is quite an art, and there are many techniques for dealing with this problem. On the other hand, these issues are separate from the basic operation of the interpreter itself.

become analogous to the machine language of a conventional Von Neumann computer. We will then learn about *compiling* controller instructions for an algorithm, for execution by the interpreter data paths.

5.1. Computing with register machines

As a first example, let us consider how to compute the greatest common divisor, using the process expressed by the SCHEME procedure that we first introduced in Chapter 1:

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

Now, just what does it take to compute GCD's by this algorithm? We have two numbers, *a* and *b*, and we will assume that we have them *stored* in two registers by those names.[2] We must be able to test if the contents of register *a* is zero. We must also be able to compute the remainder of the contents of register *a* divided by the contents of register *b*. This is itself a complex process, but assume, for the nonce, that we can go out and buy a box that computes remainders -- we will see how to fix that assumption shortly.

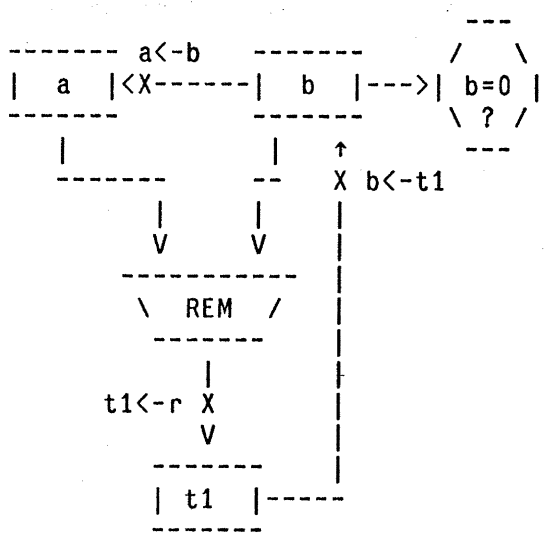
On each cycle of the algorithm, the contents of *a* is replaced by the contents of *b* and the contents of *b* is replaced by the remainder of the old contents of *a* and the old contents of *b*. It would be nice if this could be done *simultaneously*, but for simplicity here we will assume that only one register can be assigned a new value at each moment in time.[3] Thus, to avoid a *timing error* we will need another temporary register, which we will call *t1*, which will hold the value of the remainder of the old contents of *a* and *b* after *a* has been assigned to the contents of *b*.[4] We will draw this design with the following *data path* diagram:

---

[2] Of course, a number may be arbitrarily large, and any finite piece of hardware will only be able to hold a finite number, but we will avoid that complication now. We will assume that a register can be arranged to hold any number.

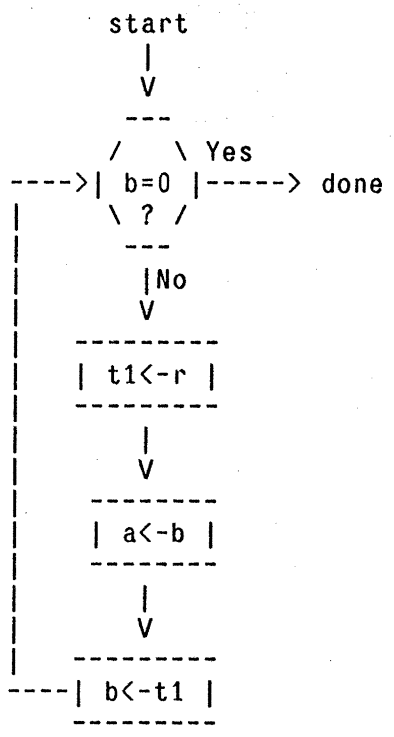
[3] Most computer design is done in terms of a catalog of parts and conventions called a *logic family*. In most logic families there are registers which have the property that they accept a new value while putting out the old one. This is arranged by having each register really be a double one, and by making the moments in time have a finite width. Such registers can be exchanged with no intermediate temporary. These *clocking disciplines* are important in simplifying real hardware design, but we will not assume any such discipline here.

[4] If you don't understand this, try the following experiment: Hold a large object in each hand, now try to interchange the objects without putting one of them down.



In this diagram, the registers are indicated by a box. Each way to assign a register a value is indicated with an arrow with an *X* behind the head. Each of these has a mnemonic label (such as *r<-t1*). The operator, *rem* computes the remainder of the contents of the registers *a* and *b* to which it is attached. We use that value by assigning it to *t1*. We can think of each of the *X*s as being a button which if pushed, allows the contents of the register pointed at to be replaced with the value from the source. In addition, the B register can report whether or not its contents is zero. We can think of the zero sensor as being connected to a lightbulb which lights up when the contents of the B register is zero.

Next, we must understand how this data path machine is controlled to produce the gcd of two numbers. The idea is to produce control signals which push the buttons in the right sequence. We normally think of this in terms of a *finite state machine*, which we draw with the following diagram:



We can think of such a diagram as a kind of maze for a marble to roll around in. When the marble rolls into a box, it pushes the button which is named in that box. When it rolls into a decision node, it looks at the light bulb named by that node and decides which way to go based on that result. We start the marble at START, with values in the registers *a* and *b*, and we wait until it gets to DONE.

We have just completely described a machine which computes *gcd*'s. We have not explained how the data gets into the registers in the first place, or how it is used when done, but we now have an entirely mechanistic notion which requires no volition or intelligence anywhere. We could imagine building such a computing engine with pinball-machine parts, for example, assuming we had a part which computes remainders.

5.1.1. Register transfer machine language

Our diagrams are adequate for describing small machines, such as *gcd*, but they rapidly become unwieldy for describing big machines, such as an evaluator. To make it possible to describe big machines, we will create a special language for compactly representing our designs.

Our language is organized around defining a compound operation on registers in terms of more primitive ones. Such a definition consists of two parts, a declaration section which describes the data paths -- the arrangement of registers and operators which we will use to perform the computation -- and a control section which describes the state machine which will be used to sequence the data paths.

First we need a means of notating operations on registers. Registers are accessed by two special expression fragments -- *fetch* and *assign*. The *fetch* fragment notates the ability

to access the contents of a register, and the *assign* fragment notates the ability to store data in a register. For example, we will notate the operation of replacing the contents of register *z* with the sum of the contents of registers *x* and *y* by the following syntactic form:

```
(assign z (+ (fetch x) (fetch y)))
```

Our *gcd* machine, described above, is designed to perform the operation *(assign a (gcd (fetch a) (fetch b)))*. Our method of computing *gcd* depends upon having available the operation *(assign t1 (remainder (fetch a) (fetch b)))*. We also must be able to make certain tests on the contents of particular registers. For example, we need to be able to test if register *b* contains a zero. We will notate such a test as the fragment *(zero? (fetch b))*.

The controller section for a machine will be described by a microprogram, which is a sequence of the operations declared in the data path section. Operations in a program sequence are assumed to be executed sequentially. Since our state machines have loops, we need a *goto* instruction to continue execution a labeled place in the text of the program. We also need a branch instruction to allow us to conditionally continue execution (based on a test declared in the data path section) at a labeled place. We notate this with an *branch* form such as *(branch (zero? (fetch b)) done)*.

Places in a program to which control may be transferred are called *entry points*. They are labeled with mnemonic names.

For example, we can write down a description of our *gcd* machine as follows:

```
(define-machine gcd
  ;; Purpose: (assign a (gcd (fetch a) (fetch b)))
  ;; Side effect is to set contents of b to zero.

  ;; First we declare our data paths.
  (registers a b t1)            ;These are the registers we may use.
  (operations                   ;These are the operations we may use.
   (assign t1 (remainder (fetch a) (fetch b)))
   (assign a (fetch b))
   (assign b (fetch t1))
   (branch (zero? (fetch b)) gcd-done)
   (goto test-b))

  ;; This is the program for the controller.
  (controller
   test-b                       ;This is a label
     (branch (zero? (fetch b)) gcd-done)
     (assign t1 (remainder (fetch a) (fetch b)))
     (assign a (fetch b))
     (assign b (fetch t1))
     (goto test-b)              ;Continue at label above
   gcd-done))
```

Despite its Lisp-like syntax, a description such as the one above is much simpler than a Lisp procedure. An instruction, in general, is either a *goto*, a branch, or the assignment to a register of the value of some operation applied to the contents of registers. In particular, no "nested procedure calls" are allowed in instructions.

5.1.2. Compound submachines

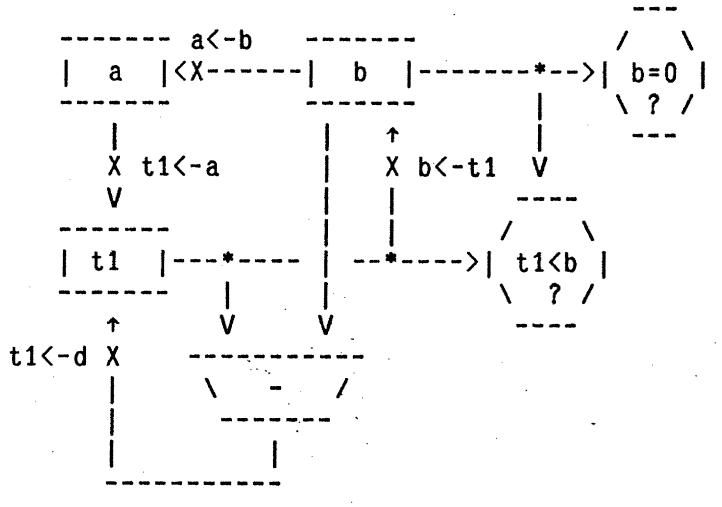
Our *gcd* machine depends upon having a method of taking the remainder of the contents of the registers *a* and *b* and putting the result into the register *t1*. This may not itself be primitive, but may be defined in terms of simpler operations like subtraction. Indeed, we may write a SCHEME program to compute remainders in this way:

```
(define (remainder n d)
  (if (< n d)
      n
      (remainder (- n d) d)))
```

If we wanted to build our *gcd* machine in terms of these simpler parts, we would want to define the operation *(assign t1 (remainder (fetch a) (fetch b)))*:

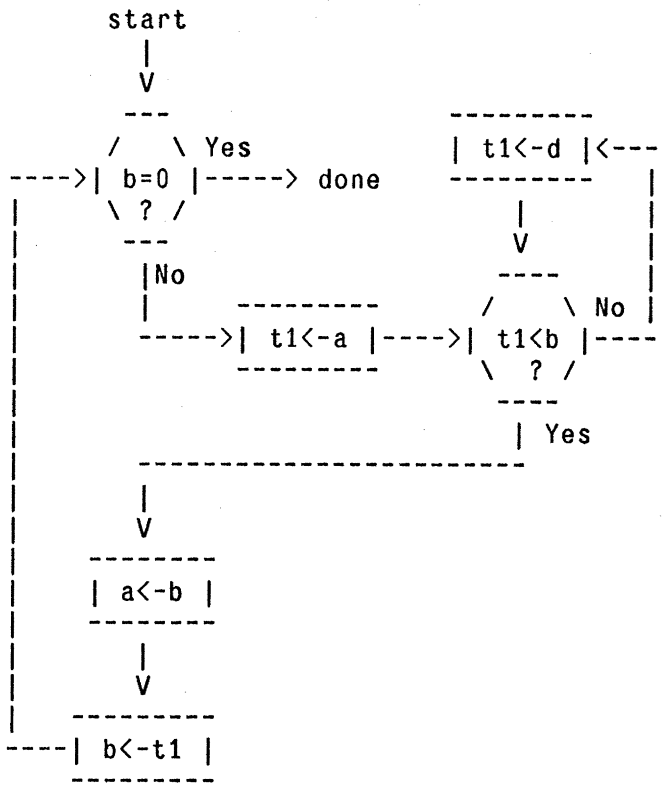
```
(define-submachine gcd
  (purpose (assign t1 (remainder (fetch a) (fetch b))))
  (registers a b t1)
  (operations
   (assign t1 (fetch a))
   (assign t1 (- (fetch t1) (fetch b)))
   (branch (< (fetch t1) (fetch b)) rem-done)
   (goto rem-loop))

  (controller
     (assign t1 (fetch a))
   rem-loop
     (branch (< (fetch t1) (fetch b)) rem-done)
     (assign t1 (- (fetch t1) (fetch b)))
     (goto rem-loop)
   rem-done))
```

We may think of this definition of the *remainder* submachine as elaborating the definition of the *gcd* machine which we defined above. It removes the *remainder* operator from the data path design, replacing it with a new operator, -, and a new test, <. (It could have added new registers, too, though this was unnecessary here. Our elaborated *gcd* machine data path now looks like:



Our more detailed design also elaborates the state machine for our *gcd* machine by replacing the state which assigns to *t1* the remainder of *a* and *b* with a sequence of states

which contains a loop:



In fact, if we had a more complex machine than *gcd* which used several instances of the operation *(assign t1 (remainder (fetch a) (fetch b)))* we imagine that each state in which the operation is called is elaborated in this way. Thus we see that defining a submachine allows a designer to explain the implementation of any particular register operation in terms of simpler operations.

5.1.3. Sharing in machines

When designing a computer to perform a particular task, it is often economical to share hardware. For example, if we want to make a machine which uses many *gcd* operations in its algorithm, we do not usually want to duplicate the *gcd* hardware for each use of it in the description of the larger machine. This problem is explicitly not addressed by the submachine construct described above. In fact, each mention of a submachine is intended to expand into a set of states in the parent state machine, and into an augmented datapath in that machine.

Subroutines and continuations

As usual, the hardware sharing problem has two aspects, sharing the data paths and sharing the controller. Consider, for a moment, only the problem of sharing the controller hardware for several instances of a sequence of states. For example, suppose we have a controller specification which has several instances of the *gcd* sequence in it:

```
(controller
```

```
    ...

    gcd-1
    (branch (zero? (fetch b)) after-gcd-1)
    (assign t1 (remainder (fetch a) (fetch b)))
    (assign a (fetch b))
    (assign b (fetch t1))
    (goto gcd-1)
    after-gcd-1

    ...

    gcd-2
    (branch (zero? (fetch b)) after-gcd-2)
    (assign t1 (remainder (fetch a) (fetch b)))
    (assign a (fetch b))
    (assign b (fetch t1))
    (goto gcd-2)
    after-gcd-2

    ...

    )
```

We want to merge these two sequences into one, but we still have two places in the code which want to do a *gcd* operation. When each of them is done we want to continue executing the state machine code after the place where the corresponding *gcd* was needed. One idea is to distinguish the two places from which the *gcd* was invoked by putting a distinguishing *token* into a special register, *gcd-continue*, which we will use to decide which sequence to return to after performing the *gcd* operation. We then have only one copy of the *gcd* routine which ends by returning to one or the other of the two invocations based on the value of the *gcd-continue* register.

```
(controller

    ...

    ;; We have only one copy of the GCD routine here.
    gcd
    (branch (zero? (fetch b)) gcd-done)
    (assign t1 (remainder (fetch a) (fetch b)))
    (assign a (fetch b))
    (assign b (fetch t1))
    (goto gcd)
    gcd-done
    (branch (zero? (fetch gcd-continue)) after-gcd-1)
    (goto after-gcd-2)

    ...

    ;; We invoke it here for the first time.
    (assign gcd-continue 0)
    (goto gcd)
    after-gcd-1

    ...
```

```
;; We invoke it here for the second time.
(assign gcd-continue 1)
(goto gcd)
after-gcd-2

...

)
```

This is a pretty good idea, but it does not easily generalize to 100 instances of *gcd* computations. In addition, each subroutine sequence will require its own register to hold continuation information. A better idea is to allocate a special continuation register which is common among all of the subroutines in a system. The continuation register will be allowed to hold an identification for a state in the system. This register will contain information about where the subroutine should continue executing when it is done.

This requires a new kind of connection between the data paths and the controller. There must be a way of identifying a state in the controller maze, so that this identifier can be placed in a register. We must also allow the state machine to take its next state specification from the continuation register rather than just from a constant. Thus there must also be a way of taking this identifier out of the register and using it to continue execution at the state so identified.

In order to notate these we must extend our state machine language. We will extend the *assign* operation fragment to allow a register to be assigned to a state label. We will also extend the *goto* operation to allow one to continue at the state described by the contents of a register. Thus we will allow the following sorts of operations:

```
(assign continue after-gcd-2)

(goto (fetch continue))
```

Using these new constructs we may now rewrite our state machine program as follows:

```
(controller

   ...

   ;; We have only one copy of the GCD routine here.
   gcd
   (branch (zero? (fetch b)) gcd-done)
   (assign t1 (remainder (fetch a) (fetch b)))
   (assign a (fetch b))
   (assign b (fetch t1))
   (goto gcd)
   gcd-done
   (goto (fetch continue))

   ...

   ;; We invoke it here for the first time.
   (assign continue after-gcd-1)
   (goto gcd)
   after-gcd-1

   ...
```

```
;; We invoke it here for the second time.
(assign continue after-gcd-2)
(goto gcd)
after-gcd-2

...

)
```

Argument registers and calling conventions

We note here that each invocation of the *gcd* subroutine gets its first argument from *a*, its second argument from *b* and develops its answer in the *a* register, modifying the values of the *b* and *t1* registers in the process. This may be convenient sometimes, but it can cause trouble if we want several disparate sequences to call *gcd*. It is useful to arrange *conventional interfaces* for commonly used subroutines, so that they can be safely used without worrying about them destroying valuable data in shared registers. One way to organize this is to establish *calling conventions* which specify those registers which a subroutine will expect to find its arguments in and the registers where it is expected to stash its value (or values, if more than one) for the continuation to find it. These *linkage* registers are then reserved for that putpose, and a program may not assume that they are free to hold any data which may be valuable, across a subroutine call. This sort of arrangement essentially decomposes a machine which contains several subroutines into relatively independent parts which are loosly coupled only through the subroutine linkage registers and the continuation register.

5.1.4. Using a stack to implement recursion

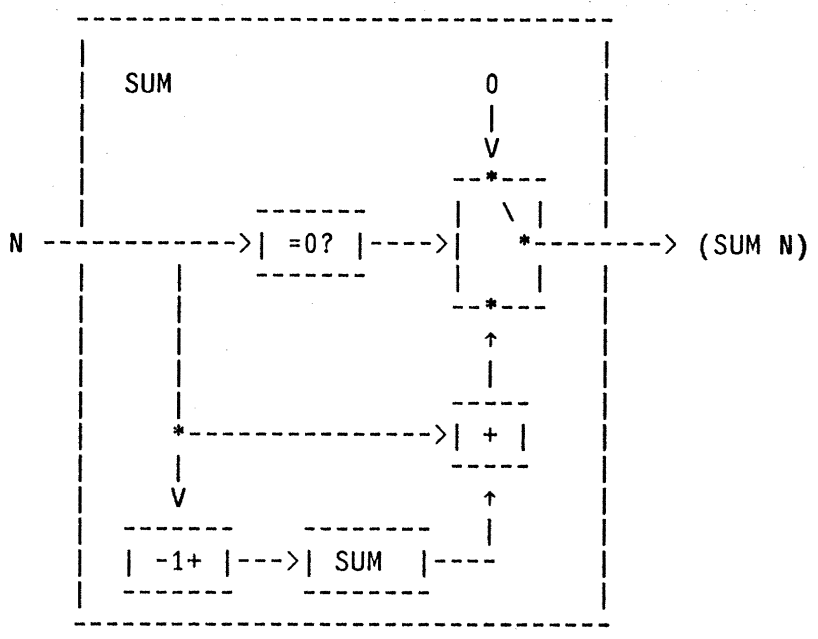
Using the ideas illustrated above, we can translate any iterative process into a register machine. We must allocate enough registers to hold all of the state variables of the process. The state of the process is determined by the contents of the registers, and the machine continually executes a program loop, changing the contents of the registers, until some termination condition is satisfied. Implementing recursive processes, however, requires an additional mechanism.

Consider the following recursive method for computing the sum of the first *n* integers:

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

If we plan to implement this procedure as a register machine, we see that computing *(sum n)* requires us, as a subproblem, to compute *(sum (-1+ n))*. On the other hand, once we have this value we are not yet done with our computation. We still must add the old value of *n* to the answer to the subproblem. This last step was not needed in an iterative problem.

We can think of the recursive sum computer as a machine, with parts which test for zero, add, decrement, switch values, produce constants, and compute sums. Because this machine contains a copy of itself inside of itself it cannot be implemented with a fixed, finite number of parts.



We must arrange an illusion, whereby we use the same hardware for each nested instance of the sum machine. Now the entire state of the sum machine is contained in its registers, and in the place at which it will continue when the subproblem is completed. Thus the illusion can be sustained by copying all of the relevant registers into a safe place. This essentially suspends the operation of the outer machine so that it can be continued later. We can then use the hardware to compute the result of the inner machine. We then restore the state of the outer machine and continue its execution.

In the case of *sum* we will need to save the old value of *n*, to be restored when we are through with computing the *sum* of the decremented *n*. In addition, the value of the subproblem must be delivered to the correct continuation of the *sum* machine.

In summary, the the general strategy for implementing an algorithm which has recursive subproblems is as follows: When a recursive subproblem is encountered one must first save the values of the registers whose current values will be required after the subproblem is solved, then solve the recursive subproblem, and then restore the saved registers and continue execution on the main problem.

Since there is no *a priori* limit on the depth of a recursive process we may need to save an arbitrary number of values. Notice also that the values will be restored in the inverse order to which they are saved, since, in a nest of recursions, the last subproblem to be entered is the first to be finished. Thus we can save register values on a *stack*, or "last-in-first-out" data structure. The stack is potentially unbounded, though any particular stack is of only finite depth. We will later see how to implement such a structure, but for now, let us just assume that it exists.

We now extend our register machine language to have a stack. Values are placed on the stack using the `save` operation, and restored from the stack using the `restore` operation. After a sequence of values has been `saved` on the stack, a sequence of `restores` will

retrieve these values in inverse order.[5] We will declare those registers which may be *saved* or *restored* as operations within the data path specification of a machine being designed.

In a recursive process, the *continue* register will be used to hold the current continuation. This will have to be *saved* and *restored* just like any other register, if its current value will be needed later and if the operations we are about to do may lose its current value.

Here is how we use this strategy in implementing the recursive *sum* procedure given above. We will define a machine with a stack and three registers, called *n*, *val*, and *continue*:

```
(define-machine sum
  ;; Purpose: (assign val (sum (fetch n)))
  (registers n val continue)
  (operations
   (assign n (-1+ (fetch n)))
   (assign val 0)                ;May specify a constant.
   (assign val (+ (fetch n) (fetch val)))
   (assign continue after-sum)
   (assign continue sum-done)
   (save continue)
   (restore continue)
   (save n)
   (restore n)
   (branch (zero? (fetch n)) zero-sum)
   (goto sum-loop)
   (goto (fetch continue)))
  (controller
     (assign continue sum-done)
   sum-loop
     (branch (zero? (fetch n)) zero-sum)
     (save continue)
     (assign continue after-sum)
     (save n)
     (assign n (-1+ (fetch n)))
     (goto sum-loop)
   after-sum
     (restore n)
     (restore continue)
     (assign val (+ (fetch n) (fetch val)))
     (goto (fetch continue))
   zero-sum
     (assign val 0)
     (goto (fetch continue))
   sum-done))
```

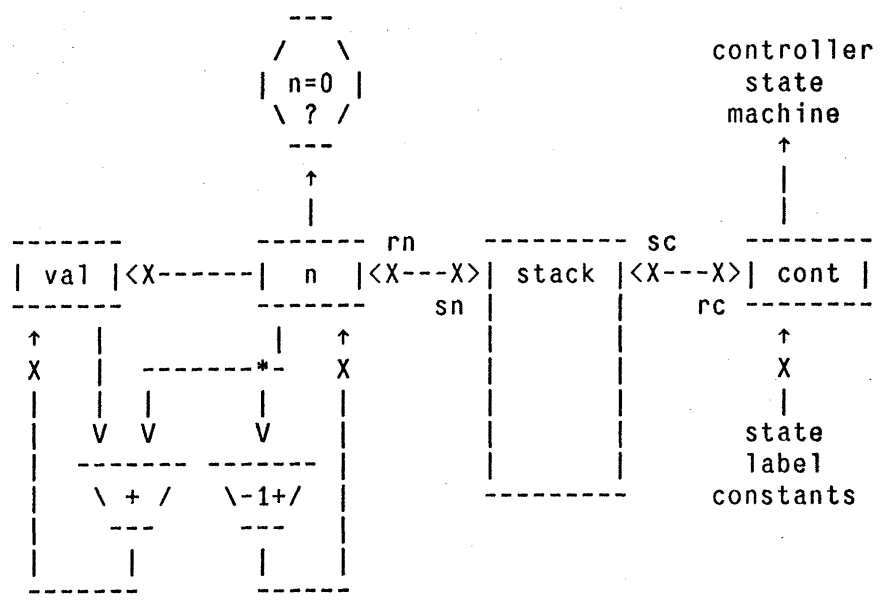
When we begin, the highest integer to be summed should be assigned to *n*. *continue* is initialized to the label *sum-done* which terminates the computation when the recursion is finished. The answer will then be in *val*. Notice that *val* is not saved in this computation because it is used for communicating the value developed from one level of computation to the next. Its old value is not useful after the subroutine returns, only its new value, the value of the subcomputation, is needed.

The diagram below is a data-path diagram for the *sum* machine. We have left out most of

---

[5] We can define a stack as a abstract data structure by specifying that *save* and *restore* satisfy the following conditions: If the stack is in state *s* and register *x* has contents *c*, then execution of *(save x)* will put the stack into state *s'*. If, when the stack is in state *s'* we execute *(restore x)*, the contents of register *x* reverts to *c* and the stack reverts to state *s*.

the assignment labels, showing only the ones for stack operations.



In the example above, we implemented a linear recursive process as a register transfer machine. In the register transfer machine, the recursion looks like an iteration. The reason is that we have hidden all of the unbounded state in the stack data structure, leaving only a finite amount of state on the surface to be manipulated directly. The key is that only one instance of the *sum* computer need be active at any one time, hence the suspended state of the deferred instances may be hidden.

A double recursion

To make sure that our use of a stack to save the state of a register so that it can be restored later, is clearly understood, let us look at a more complex recursive process. Consider, for example, the tree recursion used to compute Fibonacci numbers:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

The following description is a design for a machine which implements this fibonacci number computation:

```
(define-machine fib
  ;;Purpose: (assign val (fib (fetch n)))
  (registers n val continue)
  (operations
   (assign n (- (fetch n) 1))
   (assign n (- (fetch n) 2))
   (assign n (fetch val))
   (assign val (fetch n))
   (assign val (+ (fetch val) (fetch n)))
   (assign continue afterfib-n-1)
   (assign continue afterfib-n-2)
   (assign continue fib-done)
   (save n)
   (restore n)
   (save val)
   (restore val)
   (save continue)
   (restore continue)
   (branch (< (fetch n) 2) immed-ans)
   (goto fib-loop)
   (goto (fetch continue)))
  (controller
   (assign continue fib-done)
   fib-loop
   (branch (< (fetch n) 2) immed-ans)
   (save continue)
   (assign continue afterfib-n-1)
   (save n)                        ;Save old value of n
   (assign n (- (fetch n) 1))      ;Clobber it to n-1
   (goto fib-loop)
   afterfib-n-1                    ;VAL contains fib(n-1)
   (restore n)
   (restore continue)
   (assign n (- (fetch n) 2))      ;n now has n-2
   (save continue)
   (assign continue afterfib-n-2)
   (save val)                      ;will need fib(n-1)
   (goto fib-loop)
   afterfib-n-2
   (assign n (fetch val))          ;shuffle registers.
   (restore val)
   (restore continue)
   (assign val (+ (fetch val) (fetch n)))
   (goto (fetch continue))
   immed-ans
   (assign val (fetch n))
   (goto (fetch continue))
   fib-done))
```

Exercise 5-1: For each of the following procedures, you are to specify a register machine (describing its registers and primitive operators and tests) and a program which implements the procedure as a register machine. You may assume that machine registers are allowed to hold list data, and that the primitive list operations *cons*, *car*, *cdr*, *atom*, *eq?*, and *null?* may be declared to be primitive in your machines. You should draw simple diagrams showing the data paths and the controllers for each of the machines you implement.

a. Recursive factorial

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (-1+ n)))))
```

b. Iterative factorial

```
(define (fact n)
  (define (fact-iter count ans)
    (if (zero? count)
        ans
        (fact-iter (-1+ count) (* count ans))))
  (fact-iter 1 1))
```

c. Recursive countatoms

```
(define (countatoms tree)
  (cond ((null? tree) 0)
        ((atom? tree) 1)
        (else (+ (countatoms (car tree))
                 (countatoms (cdr tree))))))
```

d. Tail-recursive countatoms

```
(define (countatoms tree)
  (define (count-iter tree n)
    (cond ((null? tree) n)
          ((atom? tree) (1+ n))
          (else (count-iter (cdr tree)
                            (count-iter (car tree)
                                        n)))))
  (count-iter tree 0))
```

Exercise 5-2: For each of the programs *gcd*, *sum*, and *fib* in the text, "hand simulate" its behavior on some non-trivial input (requiring execution of at least one recursive call). Assuming that the stack is simulated with LISP lists, and that registers are simulated with LISP variables, as follows, show the contents of the stack at each significant point in the execution of the program.

```
(fetch x)
        ==> x
(assign x y)
        ==> (set! x y)
(save x)
        ==> (set! stack (cons x stack))
(restore x)
        ==> (sequence (set! x (car stack))
                      (set! stack (cdr stack)))
```

Exercise 5-3: Ben Bitdiddle observes that the *fib* machine control program in the text has an extra *save* and an extra *restore* which may be removed to make a better program. Where are they? Explain why they are not really needed.

5.1.5. Problem section: A register machine simulator

To get a good understanding of the design and programming of register machines, we need a way of testing register machine designs to see if they do what we expect them to do. We could go on to learn how to build actual hardware to actually implement any particular design. On the other hand, we can do many software experiments in the time required to build just one hardware gadget. We also know that we already have a universal machine, so we can simulate any desired machine design with it, if we have an appropriate simulator for that class of machine. But a simulator for a class of machines can be organized as an evaluator for the language used to describe the machines in that class.

In this section we will set up a simulator (in Scheme) for our register-transfer language. We will use it to run some of our simple programs. We will extend it and embellish it with features.

Later we will use it to perform some timing experiments on our register-transfer machine for a Scheme evaluator. This may seem a bit incestuous, but it is the usual way one does experiments to get insight into a new design or a new language.

The simulator will take a machine description and use it to set up a model of the machine to be simulated in the computer. The model will be a specially constructed data structure which will have parts which correspond to the parts of the machine to be simulated. We will be able to simulate the machine by executing programs which manipulate the model.

The first part of the simulator is the *assembler*, a program which takes the machine description and makes the model. We will assume that by some syntactic magic the expression

```
(define-machine <name>
  (registers . <regs>)
  (operations . <ops>)
  (controller . <code>))
```

is converted into a call to the assembler:

```
(define <name> (assemble '<regs> '<ops> '<code>))
```

The assembler makes up a new machine model (which has no parts specifically designed for our particular machine). It then builds the model registers, the model operations, and the model controller into our new machine model. These are performed by mutation of the machine model data structure. The new mutated machine model is returned by *assemble*.

```
(define (assemble registers operations controller)
  (let ((machine (make-new-machine)))
    (set-up-registers machine registers)
    (set-up-operations machine operations)
    (set-up-controller machine controller)
    machine)))
```

The registers are set up in the target machine by means of *remote-define!*, a mutator for machines which sets up an association between the name of the register and the data structure which is made up by *make-register* to model a register. In addition, a note is made in the model machine of the registers which were constructed. This is done by *remote-set!* which changes the value of a named preexisting slot in the machine model.

```
(define (set-up-registers machine registers)
  (remote-set! machine '*registers* registers)
  (mapc (lambda (register-name)
          (remote-define! machine register-name
                          (make-register register-name)))
        registers))
```

Next we must set up the operations defined by the designer of our machine to designate the legitimate instructions of the machine. Here we build the **instruction-map**, which is a mapping between the operation expressions which are declared by the user and the instructions which will implement those operations in the model. The magic is in the procedure *instruction*, which constructs the appropriate instruction to be executed. We will see this later.

```
(define (set-up-operations machine operations)
  (remote-set! machine '*instruction-map*
    (mapcar (lambda (operation)
              (list operation
                    (instruction machine operation)))
            operations)))
```

Finally, we must set up the model controller. The key here is to paste together a sequence of instructions which will implement the operations intended by the designer. The trick is in the disposition of labels. The labels designate certain entry points into the sequence of instructions in the controller. These are set up by *make-label*. In addition, a special entry point, called **start**, is set up to allow us to find the beginning of the controller graph.

```
(define (set-up-controller machine controller)
  (define (loop lines)
    (if (null? lines)
        '()
        (let ((rest (loop (cdr lines))))
          (if (symbol? (car lines))     ; Statement label
              (sequence (make-label machine (car lines) rest)
                        rest)
              (cons (get-instruction machine (car lines))
                    rest)))))
  (remote-set! machine '*start* (loop controller)))
```

Labels are just checked to see if they are already defined in this machine, (using the special slot called **labels** which accumulates the set of labels used in the machine.)

```
(define (make-label machine label labeled-entry)
  (let ((defined-labels (remote-get machine '*labels*)))
    (if (memq label defined-labels)
        (error "Multiply defined label" label)
        (sequence
         (remote-define! machine label labeled-entry)
         (remote-set! machine
                      '*labels*
                      (cons label defined-labels))))))
```

Also, instructions to be inserted into the model controller program are fetched from the **instruction-map** which was constructed from the operation declarations given earlier. This allows the designer to check that he uses no operations which he did not declare.

```
(define (get-instruction machine operation)
  (let ((pcell (assoc operation
                      (remote-get machine
                                  '*instruction-map*))))
    (if (null? pcell)
        (error "Undeclared operation" operation)
        (cadr pcell))))
```

The representation of the model machine

This is the entire surface level of the simulation assembler. Now we have to get down to describing how the model machine actually works, and how it is implemented!

The model register machine is represented as a Scheme environment. The registers are *cons*-cells whose *car* will be the contents of the register and whose *cdr* will be the name of the register (the *cdr* will not be used except for debugging reasons). Thus *fetch* will be modelled by *car* and assign will be modelled by *set-car!*. In the machine environment the name of each register will be bound to that register, thus simulated machine instructions can be Scheme programs which reference those variables to get at the registers. The register maker is thus just:

```
(define (make-register name)
  (cons nil name))

(define fetch car)
```

At this point we must understand *make-new-machine*. It is a simple program which does nothing but make up an environment and return it as its value. The environment is initialized with a bunch of variables and procedures which are going to be needed in every register transfer machine.

```
(define (make-new-machine)
  (make-environment
   <the initial environment's contents>))
```

The code fragments which construct the initial environment's contents start here. Any code fragments in the text from this point until further notice are to be assumed to be executed in the new environment when we make a new machine.

First we should look at how an operation may be simulated. The simplest instructions are register assignments. They do nothing more than change the value of the register to a new value and then go on to execute the next instruction -- we will explain that a bit later.

```
(define (assign register value)
  (set-car! register value)
  (normal-next-instruction))
```

Other simple operations simulate the stack. We have a special register called **the-stack** which will contain the stack pointer of our simulated machine. *Save* and *restore* are simple operations which proceed to the next instruction after doing their job.

```
(define the-stack (make-register 'the-stack))

(define (save register)
  (set-car! the-stack
            (cons (fetch register) (fetch the-stack)))
  (normal-next-instruction))

(define (restore register)
  (set-car! register (car (fetch the-stack)))
  (set-car! the-stack (cdr (fetch the-stack)))
  (normal-next-instruction))
```

The controller is represented as a sequence of instructions, each of which is a Scheme procedure which takes no arguments, and whose body is the operation which designates what it must do. It will have been defined in the machine environment so its free variables (the register names and labels) will refer to the correct values in the simulated machine.

Now the register machine must have a way of knowing where it is in executing a controller code sequence. We model this with a special register, called the *program-counter*, which always points at the beginning of the sequence of instructions to be executed. Instructions will be executed from the program counter until it runs out:

```
(define program-counter (make-register 'program-counter))

(define (execute-next-instruction)
  (cond ((null? (fetch program-counter)) 'done)
        (else
         ((car (fetch program-counter)))
         (execute-next-instruction))))
```

Each normal (non-branching) instruction calls *normal-next-instruction* to get to the next instruction. *Goto* is used to start executing at a given sequence. *Branch* is used to conditionally start execution at a given sequence.

```
(define (normal-next-instruction)
  (set-car! program-counter (cdr (fetch program-counter))))
```

```
(define (goto new-sequence)
  (set-car! program-counter new-sequence))

(define (branch predicate alternate-next)
  (if predicate
      (goto alternate-next)
      (normal-next-instruction)))
```

If it is necessary to model an operation which is not a simple register operation (such as printing on a terminal or mutating a data structure with Scheme code) we can make it into a normally sequencing operation using *perform*:

```
(define (perform operation)      ;Ignores argument which is value of
  (normal-next-instruction))     ; of operation done for effect.
```

Finally, the basic machine has special slots set aside for the attachment of documentation and debugging information.

```
(define *instruction-map* nil)
(define *labels* nil)
(define *registers* nil)
(define *start* nil)
```

This is the entire contents of the initial machine environment. At this point further program fragments are not in the body of the *make-environment* expression shown above.

Building the simulation machine environment

Now that we understand how a simulated machine is constructed as an environment, we now must see how to use it to construct a simulation. This is mostly done by relative evaluation (evaluation relative to the environment describing the machine). An operation is constructed by the assembler as a procedure of no arguments defined in the simulation environment. The assembler also must remotely access, set and define the simulation variables:

```
(define (instruction machine operation)
  (eval (list 'lambda '() operation) machine))

(define (remote-get machine variable)
  (eval variable machine))

(define (remote-set! machine variable value)
  (eval (list 'set! variable (list 'quote value))
        machine))

(define (remote-define! machine variable value)
  (eval (list 'define variable (list 'quote value))
        machine))
```

Using the simulator

Finally, we must be able to use and interact with our simulated machine to see if it works. To do so we define a few useful procedures:

```
(define (remote-fetch machine regname)
  (car (remote-get machine regname)))

(define (remote-assign machine regname value)
  (set-car! (remote-get machine regname) value))
```

```
(define (start machine)
  (eval '(goto *start*) machine)
  ((access execute-next-instruction machine)))
```

Exercise 5-4: For each of the designs you made in problem 5-1, simulate it with our simulator to verify the design and to find bugs.

Exercise 5-5: Our simulator is not very careful about whether or not the machine being defined is, in fact, a legitimate register machine. For example, it does not check that the only registers used are the ones declared, or that operations use registers in only correct ways. In this problem you are to add some error checking to the assembly process.

Add code to the assembler to check that

a. the only symbols which are used in a *fetch*, an *assign*, a *save*, or a *restore* operation are ones which are declared as registers.

b. no declared register is used except in the appropriate context of a *fetch*, an *assign*, a *save*, or a *restore* operation.

c. no operation is declared twice.

d. the targets of goto operations are all declared labels or simple fetches from registers.

e. if the target of a goto operation is a fetch from a register, it is one which has an operation declared which assigns it a declared label.

f. operations are syntactically simple and non-recursive. That is, operations may only be of the following forms:

```
<operation> ::= (ASSIGN <register> <value>)
              | (SAVE <register>) | (RESTORE <register>)
              | (<opcode> <simple-value 1> ... <simple-value n>)
<value> ::= (<function> <simple-value 1> ... <simple-value n>)
          | <simple-value>
<simple-value> ::= (FETCH <register>) | <constant>
```

Exercise 5-6: This problem is really a rather big project, which we are now prepared to accomplish. The problem is to implement submachine overlays like the one we used to expand the *remainder* part of the *gcd* machine. In this problem, you are to assume that a machine is assembled, and then a submachine is overlayed on it to implement some specific *function* or *opcode* in the machine's language. You must figure out how to scan out the places where that *opcode* or *function* is used, and patch the code to run the submachine instead.

5.2. The explicit control evaluator

We have seen how simple programs written in an expression language, like SCHEME, can be transformed into specifications for a register machine, and how the control for such a machine can be represented in terms of a register-transfer language. We will now perform the same transformation on a much more complex program -- the meta-circular interpreter that we developed to explicate the operation of the SCHEME system (see page 244).

There are several problems that we will encounter in this process. Of immediate concern is just what registers and register operations we will allow in our register-machine model. Our meta-circular interpreter was written in terms of abstractly defined syntax. Thus, for example, we wrote the *eval* dispatch using such procedures as *self-evaluating?*, *quoted?*, and *text-of-quotation*. We could easily expand these into compositions of list-structure data primitives, because they are all of the form:

```
(define (quoted? exp)
  (and (not (atom? exp))
       (eq? (car exp) 'quote)))
```

Unfortunately this would make our evaluator very long, obscuring the structure with a plethora of details. Thus, we propose, at first, to allow any simple list-data manipulation operations, such as *quoted?*, as primitives of the register machine. We will also assume, for now, that the environment constructors, selectors, and mutators of the meta-circular evaluator are available in our register machine primitives, even though they implement a complex abstract table mechanism. We could, in principle, implement these as submachines, if necessary.

But even in terms of this rather simplified and abstracted model, we will not give a complete formal definition of the evaluator machine here. For example, we will only give a complete list of the operations allowed in the evaluator data paths as an appendix after we look at the controller. In fact we will really only sketch out the machine's structure and concentrate on the details of the controller code. We will show all of the essential segments of that code so that the flow of control and the stack discipline is clear. Later, in the section 5.3, we will show how the lowest level list primitives, such as *car*, *cdr*, *cons*, *eq?*, and *atom?* are implemented in terms of a very realistic memory model.

Our Lisp evaluator machine has seven registers named *exp*, *env*, *val*, *fun*, *argl*, *unev*, and *continue*. At the beginning of the evaluation of an expression *exp* contains the expression to be evaluated and *env* contains the environment in which the evaluation is to be performed. At the end of the evaluation *val* contains the value that was developed. When the evaluator is asked to evaluate a combination, it evaluates the operator and places the result in the *fun* register. The *unev* register contains the unevaluated arguments. As the arguments in *unev* are evaluated they are accumulated in *argl*. When *argl* is complete (and *unev* is empty) the procedure in *fun* is called. The *continue* register is used as in the previous section, to hold the entry at which the evaluation should continue after completing a recursive call to the evaluator to evaluate a subexpression.

The core of the explicit control evaluator

The evaluator starts at *eval-dispatch*. This evaluates the contents of *exp* in an environment specified by the contents of *env*. After evaluation is complete, the program will return to the address stored in *continue*, and the *val* register will hold the value of the expression.

The sequence labeled with *eval-dispatch* is the central element in the evaluator. It corresponds to the *eval* procedure of the meta-circular evaluator shown on page 244. As with the meta-circular *eval*, the structure of *eval-dispatch* is a case analysis on the syntactic type of the expression being evaluated. This dispatch could have been written in a data-directed style (and in a real system probably would have been) to avoid the need to perform sequential tests, and to allow for the definition of new expression types.

```
eval-dispatch
  (branch (self-evaluating? (fetch exp)) ev-return)
  (branch (quoted? (fetch exp)) ev-quote)
  (branch (variable? (fetch exp)) ev-variable)
  (branch (definition? (fetch exp)) ev-definition)
  (branch (assignment? (fetch exp)) ev-assignment)
  (branch (lambda? (fetch exp)) ev-lambda)
  (branch (conditional? (fetch exp)) ev-cond)
  (branch (no-args? (fetch exp)) ev-no-args)
  (branch (application? (fetch exp)) ev-application)
  (goto unknown-expression-type-error)
```

Evaluating simple expressions

One slight difference between this evaluator and the meta-circular one is that we have distinguished as a new expression type expressions that are applications of procedures to no arguments. This is for organizational reasons which lead to increased efficiency. We will see more about this later.

For each type of expression, *eval-dispatch* goes to a particular *dispatch target*. Simple expressions, such as numbers, variables, quotations, and *lambda*-expressions, have no subexpressions to be evaluated, and thus they just stuff the correct value into the *val* register and continue wherever the subexpression was evaluated from. Remember that we assumed that our "machine language" includes selectors and constructors such as *operator*, *make-procedure*, *variable?*, and so on. These can be implemented by translating the equivalent Lisp procedures given in section 4.1.2. In addition, we are assuming that we may use selectors, constructors, and mutators for environment structures. Evaluation of all simple expresssions are performed by the following code:

```
ev-return
  (assign val (fetch exp))
  (goto (fetch continue))
ev-quote
  (assign val (text-of-quotation (fetch exp)))
  (goto (fetch continue))
ev-variable
  (assign val
          (lookup-variable-value (fetch exp)
                                 (fetch env)))
  (goto (fetch continue))
ev-lambda
  (assign val
          (make-procedure (fetch exp)
                          (fetch env)))
  (goto (fetch continue))
```

Evaluating combinations

The essence of our evaluator is in the evaluation of combinations. A combination is an expression which notates the application of an operator to operands. The operator is a subexpression whose value will be a procedure and the operands are subexpressions whose values will be the arguments of the procedure.

We start with the case of a singleton combination -- an operator with no operands. This should be evaluated to a procedure which will be applied to no arguments. First the evaluator must evaluate the operator part of the expression. The rest of the expression is useless and will not be saved. The evaluation of the subexpression is accomplished by moving the

subexpression into the *exp* register and going off to *eval-dispatch*. The environment is correct at this point. It is not saved because we will not have to use it to evaluate any other parts of the expression before going to apply the procedure. On the other hand, the continuation must be saved because we will need it for evaluating the body of the procedure later. The new continuation for the evaluation of the operator is *apply-no-args*.

```
ev-no-args
  (assign exp (operator (fetch exp)))
  (save continue)
  (assign continue apply-no-args)
  (goto eval-dispatch)
```

When the operator expression of a singleton combination has been evaluated, execution continues at *apply-no-args* with the value of the operator in *val*. This value (which is hopefully an applicable procedure object) must be stashed into the *fun* register and the list of arguments must be set up to be empty so that we can *apply* this procedure to its arguments. We must remember that we have the continuation for the combination currently *saved*. We will have to restore it before we return the value of the procedure.

```
apply-no-args
  (assign fun (fetch val))
  (assign argl '())
  (goto apply-dispatch)
```

The operand evaluation loop

Next we must consider the general case of a combination. Here, in addition to the operator expression to be evaluated to get the procedure, there are also operands to be evaluated *in the same environment* to get the arguments. Thus, when we go off to evaluate the operator, below, we also set up the *unev* register to contain the unevaluated operand parts of the expression. We must *save* the *unev* and *env* registers across the evaluation of the operator. When we finish the evaluation of the operator, execution is to continue at *eval-args*.

```
ev-application
  (assign unev (operands (fetch exp)))
  (assign exp (operator (fetch exp)))
  (save continue)
  (save env)
  (save unev)
  (assign continue eval-args)
  (goto eval-dispatch)
```

When we have returned from evaluating the operator subexpression of a general application, we must go on to evaluate the operands and accumulate the resulting arguments in an argument list. First we restore the unevaluated operands and the environment (which we will need presently. Then we stash the procedure which was the value of the operator in *fun*, and *save* it. Finally, we initialize an empty argument list which we will use to accumulate the arguments. We then start the argument evaluation loop.

```
eval-args
  (restore unev)
  (restore env)
  (assign fun (fetch val))
  (save fun)
  (assign argl nil)
  (goto eval-arg)
```

The argument evaluation loop has two phases. In the first phase we set up to evaluate an

operand. We save up the accumulated arguments (in *argl*), the environment, and the unevaluated expression fragment. We set the *exp* register to the operand to be evaluated, and we go off, returning to the accumulation phase. A special case is made for the evaluation of the last operand because its evaluation does not require saving the environment or the list of unevaluated operands as they will no longer be needed after the last operand is evaluated.

```
eval-arg
  (save argl)
  (assign exp (first-operand (fetch unev)))
  (branch (last-operand? (fetch unev)) eval-last-arg)
  (save env)
  (save unev)
  (assign continue accumulate-arg)
  (goto eval-dispatch)
```

When the operand is evaluated, it must be accumulated onto the *argl*. The (now evaluated) operand is then removed from the list of unevaluated operands.

```
accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (cons (fetch val) (fetch argl)))
  (assign unev (rest-operands (fetch unev)))
  (goto eval-arg)
```

The last operand differs from the previous ones in that there will be no need for the list of unevaluated operands or for the environment after the last argument is developed. Thus, we need only return to a place which restores the argument list, adds the new argument, restores the saved procedure, and goes off to *apply* it.

```
eval-last-arg
  (assign continue accumulate-last-arg)
  (goto eval-dispatch)
accumulate-last-arg
  (restore argl)
  (assign argl (cons (fetch val) (fetch argl)))
  (restore fun)
  (goto apply-dispatch)
```

Procedure application

The entry *apply-dispatch* corresponds to the *apply* procedure of the meta-circular evaluator (page 245). By the time we get to *apply-dispatch*, *fun* contains the procedure to apply and *argl* contains the list of evaluated arguments to which it must be applied (reversed in order from how they appeared in the application). As with the metacircular *apply* there are two cases to consider. Either a procedure is a primitive, in which case we pass the buck to *apply-primitive-procedure* (which we assume is accessible in our "machine language"), or it is a compound procedure (the result of evaluating a lambda expression).

```
apply-dispatch
  (branch (primitive-procedure? (fetch fun)) primitive-apply)
  (branch (compound-procedure? (fetch fun)) compound-apply)
  (goto unknown-procedure-type-error)
```

Apply-primitive-procedure holds the magical linkage between the interpreter and the details of the implementation of primitive procedures such as addition. What you should imagine going on here is that the identifier of the primitive procedure, stored in *fun* is used to dispatch to a sequence of instructions which implements the primitive. The primitive will

access its arguments from *argl* and it will stash its value in *val*. Note that here we must restore the continuation so that when we do the *(goto (fetch continue))* to return to the caller, it will be correctly set up to be the continuation of the expression which reduced to the application of the primitive procedure to its arguments.

```
primitive-apply
  (assign val
          (apply-primitive-procedure (fetch fun)
                                     (fetch argl)))
  (restore continue)
  (goto (fetch continue))
```

One minor point to note is that the definition of *apply-primitive-procedure* given for the meta-circular evaluator in section 4.1.4 will not work correctly here because the *argl* is reversed here. The patches required to make them work are simple and will not be shown.

The application of a compound procedure is very simple. Evaluation must proceed in the environment where the formal parameters of the procedure are bound to the arguments in *argl*, built upon the environment carried by the procedure. The body of the procedure, to be evaluated in this environment, is a sequence of expressions, and this is handled at the entry *eval-action-sequence* (See below).

Compound-apply is the only place in the interpreter where the *env* register is ever assigned to a new value. The new value is composed from the environment of definition of the procedure and the argument list. From the point of view of the interpreter program this is a simple data manipulation, as the environment of definition of the procedure can be selected from the procedure. (Remember that the procedure was constructed from the text and the environment in the evaluation of a *lambda* expression.)

```
compound-apply
  (assign env (make-bindings (fetch fun) (fetch argl)))
  (assign unev (procedure-body (fetch fun)))
  (goto eval-action-sequence)
```

Note that *Eval-action-sequence* will expect its continuation to be saved, thus it will do the required *restore*. It also expects the list of expressions to be evaluated to be in *unev* and the environment for the evaluation to be set up in *env*.

Also notice that *make-bindings* is just a simple data operation on the contents of the indicated registers:

```
(define (make-bindings proc args)
  (extend-environment (parameters proc)
                      arg
                      (procedure-environment proc)))
```

Evaluating sequences of expressions

When we want to evaluate a sequence of expressions, one after another, as in the body of a procedure or in the action sequence of a conditional, we let *eval-action-sequence* do the work. This entry, together with *eval-action-sequence-continue*, forms a loop that evaluates each expression in the sequence in turn. The list of as yet unevaluated expressions is kept in *unev*.

The first expression is retrieved, and if there are no more, then we go off to *last-exp* where the continuation is *restore*d and the expression is evaluated with that continuation. If there is more than one expression in the action sequence, then we must be able to evaluate

the rest of the sequence with the current environment, so we save the list of expressions and the environment, and set the continuation to be a place which will restore them, and *cdr* the *unev*, and then continue evaluating the elements of the action sequence.

```
eval-action-sequence
  (assign exp (first-exp (fetch unev)))
  (branch (last-exp? (fetch unev)) last-exp)
  (save unev)
  (save env)
  (assign continue eval-action-sequence-continue)
  (goto eval-dispatch)
eval-action-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (rest-exps (fetch unev)))
  (goto eval-action-sequence)
last-exp
  (restore continue)
  (goto eval-dispatch)
```

Tail Recursion

Recall that in Chapter 1, we said that the process described by a procedure such as

```
(define (sqrt-iter guess radicand)
  (cond ((good-enough? guess radicand) guess)
        (else (sqrt-iter (improve guess radicand)
                         radicand))))
```

is an *iterative process*. Even though the procedure is syntactically recursive -- defined in terms of itself -- is is not logically necessary for an evaluator to accumulate information to be saved in passing from one call to `sqrt-iter` to the next. An evaluator that is able to avoid saving unnecessary information on such a procedure call, and is thus able to execute a procedure like `sqrt-iter` without using up more and more storage as `sqrt-iter` continues to call itself, is called a *tail recursive* evaluator.

We cannot determine whether the the meta-circular evaluator is tail-recursive, because that evaluator inherits its mechanism for saving state from the way this is accomplished in the underlying Lisp. But in the explicit control evaluator we can explicitly trace through the evaluation process to see whether the nested call in a procedure such as `sqrt-iter` causes a net accumulation of information on the stack.

In fact, our evaluator is tail-recursive, and the reason this happens is that, on evaluating the final expression of a sequence, `eval-action-sequence` goes to *`eval-dispatch` with nothing saved on the stack*. Nothing need be saved prior to evaluating the final expression in a sequence, since the result of the sequence is determined by the result of the final expression. Hence, evaluating the final expression in a sequence, even if it is a procedure call (as in `sqrt-iter`) will not cause any information to be accumulated on the stack.

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-action-sequence` as

```
eval-action-sequence
  (branch (no-more-exps? (fetch unev)) end-sequence)    ;***
  (assign exp (first-exp (fetch unev)))                 ;***
  (save unev)
  (save env)
  (assign continue eval-action-sequence-continue)
  (goto eval-dispatch)
eval-action-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (rest-exps (fetch unev)))
  (goto eval-action-sequence)
end-sequence
  (restore continue)
  (goto (fetch continue))
```

This may seem like a minor change to our previous code for evaluation of a sequence -- only the lines commented with "***" are changed -- and our interpreter will give the same results for any expression, but this change is fatal to the tail-recursive implementation. Thus procedures like *sqrt-iter* will in fact run in space proportional to the number of iterations required to converge rather than in constant space, as in our original code. This difference can be significant. With tail recursion, an "infinite loop" can be expressed using only the procedure call mechanism, for example:

```
(define (count n)
  (print n)
  (count (1+ n)))
```

Without tail recursion, such a procedure would eventually run out of (stack) space, and expressing a true iteration would require some control mechanism other than procedure call.

Our implementation of tail-recursion in *eval-sequence* is not entirely original -- it is one variety of the well-known compile-time optimization that a procedure call immediately preceding a procedure return in code is equivalent to an unconditional transfer to the procedure entry point.

Conditionals and other kinds of expressions

A conditional expression is special form with subexpressions. The problem is to evaluate the predicate part of the first clause of a conditional and then to make a decision, based on the value of that predicate, whether to evaluate the action sequence of that clause, or whether to go on to consider the next clause. The first thing we do when encountering a conditional is to stash away the *continue* register because we will need it later to return to the evaluation of the expression which is waiting for the value of the conditional. Internal to the conditional evaluation, the continuation will be *evcond-decide* -- that piece of code which receives the value of the predicate and will make the decision. We also set up the *unev* register to be the list of pending clauses. We then fall into the code which evaluates the predicate of the first clause of the conditional. Unless the clause is an *else* clause, we must save the environment and the list of pending clauses (in *unev*) and just go off to evaluate the predicate.

```
ev-cond
  (save continue)
  (assign continue evcond-decide)
  (assign unev (clauses (fetch exp)))
evcond-pred
  (assign exp (first-clause (fetch unev)))
  (branch (else-clause? (fetch exp)) else-clause)
  (save env)
  (save unev)
  (assign exp (predicate (fetch exp)))
  (goto eval-dispatch)
```

The evaluation of the predicate part of a clause will return to *evcond-decide* shown below. It restores the list of clauses and the environment (both of which will be needed whether or not the predicate of the first clause was true). It then determines whether the predicate was true. If so it sets up for evaluating the action sequence of that clause, and if not, it *cdr*s the first clause off of the *unev* list and then goes around to *evcond-pred* to examine the next clause.

If a predicate is found to be true, or an *else* clause is found, the associated actions must be evaluated sequentially and the value of the last used as the value of the conditional expression. By the time we get to *eval-action-sequence* the list of unevaluated expressions to be evaluated as the action sequence is stashed in the *unev* register. The environment is in the *env* register, but the continuation for the conditional is not in *continue* -- it is still *save*d on the stack -- as required for *eval-action-sequence*.

```
evcond-decide
  (restore unev)
  (restore env)
  (branch (true? (fetch val)) true-predicate)
  (assign unev (rest-clauses (fetch unev)))
  (goto evcond-pred)
true-predicate
  (assign exp (first-clause (fetch unev)))
else-clause
  (assign unev (action-sequence (fetch exp)))
  (goto eval-action-sequence)
```

Assignments are handled by *ev-assignment*, which is reached from *eval-dispatch* with the indicated assignment expression in *exp*. *Ev-assignment* first evaluates the value part of the expression, and then calls an appropriate environment maintaining procedure, with the symbol and the value to be assignment, to actually install the new value in the environment.

```
ev-assignment
  (assign unev (assignment-variable (fetch exp)))
  (save unev)
  (assign exp (assignment-value (fetch exp)))
  (save env)
  (save continue)
  (assign continue ev-assignment-1)
  (goto eval-dispatch)
ev-assignment-1
 (restore continue)
 (restore env)
 (restore unev)
 (perform (set-variable-value! (fetch unev) (fetch val) (fetch env)))
 (goto (fetch continue))
```

Definitions are handled similarly.

```
ev-definition
  (assign unev (definition-variable (fetch exp)))
  (save unev)
  (assign exp (definition-value (fetch exp)))
  (save env)
  (save continue)
  (assign continue ev-definition-1)
  (goto eval-dispatch)
ev-definition-1
  (restore continue)
  (restore env)
  (restore exp)
  (perform (define-variable! (fetch unev) (fetch val) (fetch env)))
  (goto (fetch continue))
```

Finally, we get to the driver-loop of the controller for the interpreter. This is the beginning of the controller. We start by clearing out the stack, next we setup a continuation for the *read-eval-print* loop. We also initialize *val* to have a printable initial value. We then print it, read the next expression, and initialize the environment from the global environment, before going to *eval-dispatch*. When we are done evaluating, we will return to the print.

```
(controller
  (perform (initialize-stack))
  (assign continue done)
  (assign val 'scheme-register-machine-simulator)
done
  (perform (user-print (fetch val)))
  (assign exp (prompting-read '**==>))
  (assign env the-global-environment)
  (goto eval-dispatch)
```

The operations used

Here we supply the complete set of operations which are needed to implement the data paths of the evaluator engine.

```
(operations
 (assign argl nil)
 (assign argl (cons (fetch val) (fetch argl)))
 (assign continue accumulate-arg)
 (assign continue accumulate-last-arg)
 (assign continue apply-no-args)
 (assign continue done)
 (assign continue ev-assignment-1)
 (assign continue ev-definition-1)
 (assign continue eval-action-sequence-continue)
 (assign continue eval-args)
 (assign continue evcond-decide)
 (assign env (make-bindings (fetch fun) (fetch argl)))
 (assign env the-global-environment)
 (assign exp (assignment-value (fetch exp)))
 (assign exp (definition-value (fetch exp)))
 (assign exp (first-clause (fetch unev)))
 (assign exp (first-exp (fetch unev)))
 (assign exp (first-operand (fetch unev)))
 (assign exp (operator (fetch exp)))
 (assign exp (predicate (fetch exp)))
 (assign exp (prompting-read '**==>))
 (assign exp (transform-let (fetch exp)))
 (assign fun (fetch val))
 (assign unev (action-sequence (fetch exp)))
 (assign unev (assignment-variable (fetch exp)))
 (assign unev (clauses (fetch exp)))
 (assign unev (definition-variable (fetch exp)))
 (assign unev (operands (fetch exp)))
 (assign unev (procedure-body (fetch fun)))
 (assign unev (rest-clauses (fetch unev)))
 (assign unev (rest-exps (fetch unev)))
 (assign unev (rest-operands (fetch unev)))
 (assign val (apply-primitive-procedure (fetch fun) (fetch argl)))
 (assign val (fetch exp))
 (assign val (lookup-variable-value (fetch exp) (fetch env)))
 (assign val (make-procedure (fetch exp) (fetch env)))
 (assign val (text-of-quotation (fetch exp)))
 (assign val 'scheme-register-machine-simulator)

 (goto apply-dispatch)
 (goto eval-action-sequence)
 (goto eval-arg)
 (goto eval-dispatch)
 (goto evcond-pred)
 (goto unknown-expression-type-error)
 (goto unknown-procedure-type-error)
 (goto (fetch continue))
```

```
(branch (application? (fetch exp)) ev-application)
(branch (assignment? (fetch exp)) ev-assignment)
(branch (compound-procedure? (fetch fun)) compound-apply)
(branch (conditional? (fetch exp)) ev-cond)
(branch (definition? (fetch exp)) ev-definition)
(branch (else-clause? (fetch exp)) else-clause)
(branch (lambda? (fetch exp)) ev-lambda)
(branch (last-exp? (fetch unev)) last-exp)
(branch (last-operand? (fetch unev)) eval-last-arg)
(branch (let? (fetch exp)) ev-let)
(branch (no-args? (fetch exp)) ev-no-args)
(branch (primitive-procedure? (fetch fun)) primitive-apply)
(branch (quoted? (fetch exp)) ev-quote)
(branch (self-evaluating? (fetch exp)) ev-return)
(branch (true? (fetch val)) true-predicate)
(branch (variable? (fetch exp)) ev-variable)

(perform (define-variable! (fetch unev) (fetch val) (fetch env)))
(perform (initialize-stack))
(perform (set-variable-value! (fetch unev) (fetch val) (fetch env)))
(perform (user-print (fetch val)))

(save argl) (restore argl)
(save continue) (restore continue)
(save env) (restore env)
(save fun) (restore fun)
(save unev) (restore unev))
```

5.2.1. Problem section: Performance Analysis of the Evaluator

One important factor in the performance of an evaluator is how efficiently it uses the stack. This problem section assignment concerns the use of the stack by our model evaluator. We will instrument our evaluator, installing a "meter" that measures the number of stack operations used in evaluating expressions. We will do this by placing features in our stack abstraction to keep track of the number of stack operations and the maximum depth reached by the stack.

To be able to perform these exercises we will assume that you have access to the complete code of model evaluator machine described in this chapter. We will need these later in the chapter. Listings and instructions for use of this code should also be available. We will also assume that you have access to the stack monitoring code we will describe and use here.

A word of advice. You will using an evaluator that is implemented by the low-level "register machine" which is itself being simulated by a Scheme program. This multiple interpretation makes the evaluator run extremely slowly.

The monitored stack

Make-stack is a procedure which makes a message-accepting stack. The stack accepts *push*, *pop*, *initialize*, and *statistics* messages. It holds its state in various local variables.

```
(define (make-stack)
  (let ((s nil) (number-pushes nil) (max-depth nil))
    (lambda (message)
      (cond ((eq? message 'push)
             (lambda (newtop)
               (set! s (cons newtop s))
               (set! number-pushes (1+ number-pushes))
               (set! max-depth (max (length s) max-depth))))
            ((eq? message 'pop)
             (let ((top (car s)))
               (set! s (cdr s))
               top))
            ((eq? message 'initialize)
             (set! s nil)
             (set! number-pushes 0)
             (set! max-depth 0))
            ((eq? message 'statistics)
             (print (list 'total-pushes number-pushes))
             (princ (list 'maximum-depth max-depth)))
            (else
             (error "Unknown request -- STACK" message))))))
```

We define *save* and *restore* in terms of this abstraction:

```
(define (save reg)
  ((the-stack 'push) (fetch reg))
  (normal-next-instruction))

(define (restore reg)
  (assign reg (the-stack 'pop)))
```

Exercise 5-7: How does this stack work? Describe how the stack implementation does its job by going through an example of executing each type of operation it supports.

Note that the driver-loop of our model evaluator reinitializes the stack on each interaction, so that the statistics printed will refer only to stack operations used to evaluate the previous expression.

Iteration and Recursion

One of the more interesting points about the Scheme interpreter is that way that it executes "tail recursive" procedures such as

```
(define (ifact n)
  (define (iter count answer)
    (cond ((> count n) answer)
          (else (iter (+ count 1) (* count answer)))))
  (iter 1 1))
```

Even though the procedure *iter* is syntactically recursive, the evaluator executes this as an iterative process, that is, the amount of space required to compute *(ifact n)* is independent of *n*. Since the evaluator's "space" is allocated on the stack, the space required can be measured as the maximum depth reached by the stack during the evaluation.

Exercise 5-8: Start the evaluator and type in the definitions of *ifact* above. Try it out for various small

numbers such as 2, 3, and 4. Record the maximum depth and number of pushes required to compute *ifact* for each of these numbers.

1. You will find that the maximum depth is independent of *n*. What is that number?
2. Induce from your data a formula in terms of *n* for the total number of push operations used in evaluating *(ifact n)* for any $n > 1$. To do this, note that the number of operations used will be a linear function of *n*, and thus it is determined by two constants.

Exercise 5-9: For comparison with the iterative version, define the following recursive version of factorial:

```
(define (rfact n)
  (cond ((< n 2) 1)
        (else (* n (rfact (- n 1))))))
```

By running this procedure 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 *(rfact n)* for $n > 2$. (Again, these functions will be linear.)

Exercise 5-10: Summarize your experiments by filling in the following table, with entries which are appropriate expressions in *n*:

	max depth	number of pushes
recursive factorial		
iterative factorial		

Note that "max depth" is essentially the space used by the evaluator in carrying out the computation, and "number of pushes" should correlate well with the time required.

Exercise 5-11: Now monitor the stack operations in the double recursive Fibonacci computation:

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

(Don't try values of *n* larger than, say, 6, unless you are prepared to wait a very long time.)

If you take a few data points, you should be able to answer the following questions:

1. Give a formula in terms of *n* for the maximum depth of the stack required to compute *(fib n)* for $n > 2$. Hint: Remember we said in Chapter 1 that the space used by this process is linear.
2. What can you say about the total number of saves used to compute *(fib n)* for $n > 2$? Recall that the number of saves corresponds to the time used, and hence should grow exponentially. Here is a hint: Let S(*n*) be the number of saves 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. Even better, you should be able to use your formula to express S(*n*) in terms of the Fibonacci numbers.

Exercise 5-12: In these problems we have identified the space taken to run a program with the maximum depth of the stack. One might truly object that perhaps we have not really accounted for all of the space. Are we sure that in the interpretation of *ifact*, for example, we have not locked up an unbounded amount of space in the environment structures?

1. Argue that the depth of the environments in a statically scoped language is bounded by a constant. It is not the depth of calls but rather the depth in the lexical structure of the procedure.
2. Illustrate this truth by describing how to build further instrumentation, analogous to the kind we provided for you in the monitored stacks, which monitors the construction of environment

structures.

5.3. Storage allocation and garbage collection

In the evaluators that we have seen all primitive operators are either aliases for simple combinations of primitive list operations, or they are simply implemented in hardware, like addition, or, like stack operations, they can be implemented in terms of primitive list operations. We have construed the LISP data manipulation primitives to be primitive operations of the computer. This is a good abstraction when we are studying the process of interpretation, but it is not a realistic view of the way in which actual computers work. Commercially available memory systems are not organized as sets of linked list cells, but rather as linearly indexed arrays. Additionally, commercially available memories are available only in finite sizes (more's the pity). Now the free and wasteful throw-away use of data objects would cause no problem if infinite memory were available, but within a realm of finite memory it is an ecological disaster. LISP systems, which provide the user with automatic storage allocation, must support the illusion of an infinitely large list structured memory. This requires that memory which was allocated for the construction of data objects that are no longer needed be recycled for the construction of new data objects.

5.3.1. Memory as vectors

In order to model the actions that can be taken in real machine memories we introduce a new kind of data structure, called a *vector*. Abstractly, we define vectors as objects which can be quickly accessed by using a numerical index. Thus, for example, if *a* is a vector, then *(vector-ref a 5)* gets the fifth entry in the vector. We will not need to make any vectors in our discussion (since we will use them to model the fixed memory of the computer, which is not dynamically constructed) however we will need to be able to mutate a vector. To change the value of the fifth entry of the vector *a* to 7 we incant *(vector-set! a 5 7)*. Thus vectors have two primitive operators of interest to us in this discussion:

```
(vector-ref <vector> <index>)
   gets the <index>th element of the vector <vector>.

(vector-set! <vector> <index> <value>)
   sets the <index>th element of the vector <vector> to <value>.
```

What is special about vectors is that we should think of these operations as being very fast, constant time, operations. This is by contrast to finding the nth element of a list, which requires n-1 cdr operations to get to the right place in the list.

Given that we can assume the existence of vectors, we can begin to see how one might embed a list-structure memory in such a linear structure. Let us imagine, for the moment, that we have an infinite memory which is divided into two vectors -- stored in registers *the-cars* and *the-cdrs*. We will fix the infinity later. We will encode list structure as follows: A pointer to a CONS-cell is an *index* into the two vectors which constitute our memory. The CAR of the CONS-cell is the entry in *the-cars* with the index of our CONS-cell. The CDR of the CONS-cell is the entry in *the-cdrs* with the index of our CONS-cell. *Cons* is performed by allocating a new, never used index, and clobbering *the-cars* and *the-cdrs* of that index to

be the arguments to *cons*. It is easy to allocate new indices into an infinite vector, since we can count -- we presume that there is a special register, *free*, which always holds the pointer to the next available free index into our memory.

Thus we can implement the primitive list structure operations in terms of our vector memory as follows, we replace each "primitive" register operation with one or more primitive vector operations:

```
(assign <reg1> (car (fetch <reg2>)))
 = (assign <reg1> (vector-ref (fetch the-cars) (fetch <reg2>)))

(assign <reg1> (cdr (fetch <reg2>)))
 = (assign <reg1> (vector-ref (fetch the-cdrs) (fetch <reg2>)))

(set-car! (fetch <reg1>) (fetch <reg2>))
 = (perform (vector-set! (fetch the-cars) (fetch <reg1>) (fetch <reg2>)))

(set-cdr! (fetch <reg1>) (fetch <reg2>))
 = (perform (vector-set! (fetch the-cdrs) (fetch <reg1>) (fetch <reg2>)))

(assign <reg1> (cons (fetch <reg2>) (fetch <reg3>)))
 = (perform (vector-set! (fetch the-cars) (fetch free) (fetch <reg2>)))
   (perform (vector-set! (fetch the-cdrs) (fetch free) (fetch <reg3>)))
   (assign <reg1> (fetch free))
   (assign free (1+ (fetch free)))
```

Stack operations are no problem since we can model them as list operations which can themselves be implemented in terms of register and memory operations, as explained above.

```
(save <reg>)
 = (assign stack (cons (fetch <reg>) (fetch stack)))

(restore <reg>)
 = (assign <reg> (car (fetch <stack>)))
   (assign <stack> (cdr (fetch <stack>)))
```

These could be further expanded into our memory manipulations, completing our implementation.

This is almost all there is to building list structure. The only complication is the question about how non-list structure (such as numbers) are represented. Symbols are usually not a problem, as they are usually compound data structures, made of list structure, which have no special representation. The only things interesting about symbols is that the reader and printer know about the character strings which they are represented by in printed text. One easy but not very good answer to this is that negative numbers may not be legitimate indices into the memory vectors. In such a scheme all non-list-structure objects must be encoded into the negative machine numbers.

5.3.2. Supporting the illusion of infinite memory

In reality, all current memories are of finite size, so eventually we must run out of free space

in which to *cons*.[6] In any case, we are in considerable luck. Most of memory which is used in performing a computation is only constructed to hold intermediate results. These results are accessed and the cells allocated to contain them are no longer needed -- they are "garbage". For example, if we perform the computation `(accumulate + 0 (filter odd? (enumerate-interval 0 n)))` we will eventually cons-up two lists, the enumeration and the result of filtering the enumeration. On the other hand, when the accumulation is done, these lists are no longer needed and can be re-used. In addition, if we used streams to perform this process, we can reclaim most of that memory much earlier -- we need not finish enumerating before we begin to filter.

Our illusion of infinite memory can thus be arranged if we collect all of the garbage periodically, and this turns out to be about as much as we *cons*. The storage allocator must have means to reclaim memory which has been allocated but which can no longer influence the future of the computation. The problem, therefore, is to determine which parts of memory are garbage and which are still needed. Many strategies have been developed for this.[7] The one which we will see here is called "garbage collection". It is based on the observation that, at any moment in a LISP interpretation, the future of the computation is completely determined by objects which can be reached by a sequence of primitive data access operations starting with a few special data structures, such as the registers and stacks, maintained by the interpreter. This is because, in order to affect a computation, a quantity must eventually end up in an interpreter register. However, the interpreter only accesses quantities using the list selectors. Thus there is no way a quantity which is not so reachable can affect the future of a computation. Thus any memory cell which is not so accessible may be recycled.

There are many ways to accomplish garbage collection. One plan is the mark-sweep method. In this plan we recursively trace the structure pointed at by the machine registers, marking each cell reached as we go. Eventually we mark the entire transitive closure of the list access operations starting with the machine registers. Thus a cell is marked if and only if it is accessible. We then scan all of memory, and any location which is unmarked is swept up as garbage and made reusable. Another plan is the copy and compact method. It depends upon not having actually used up all of memory but only about half of it. The idea is that we recursively trace all of the structure pointed at by the machine registers (the useful, accessible data) and copy it into an auxilliary memory compactly. The memory copied from

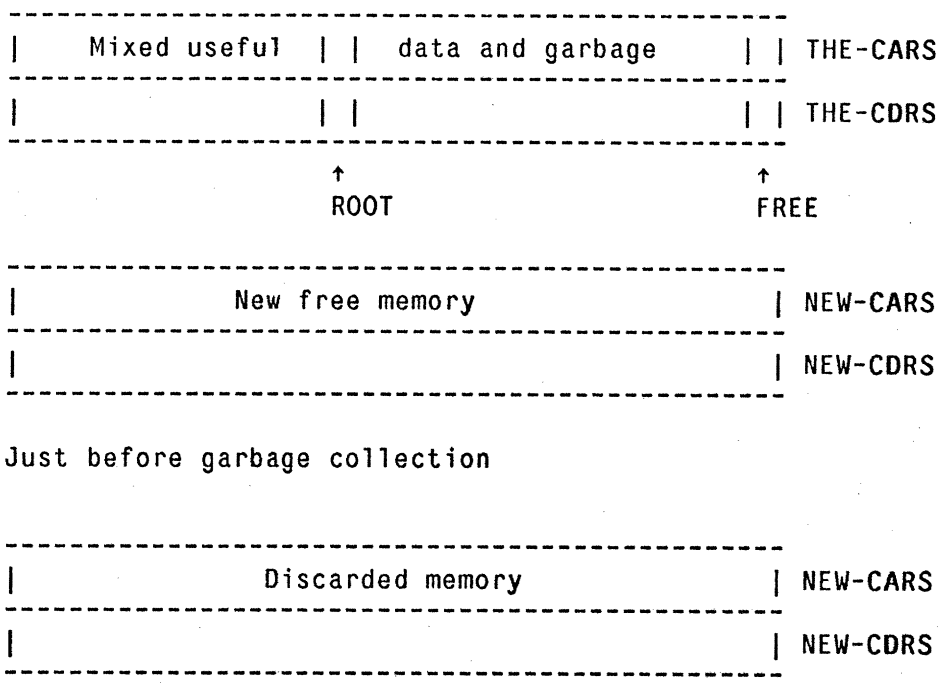
---

[6] This may not be true eventually. Memory may get cheap enough so that it is basically impossible to run out of free memory in the physical lifetime of the computer. For example, there are about pi*10↑7 seconds in a year. Each has 10↑6 microseconds. Thus, if we could *cons* once per microsecond, we would only need 10↑15 cells of memory to make a machine which would be able to run for 100 years without running out. That is only completely absurd, but not physically impossible. On the other hand, machines are getting faster, and this kind of argument will not work if we have large aggregates of parallel machines, so it is hard to say what the actual story is.

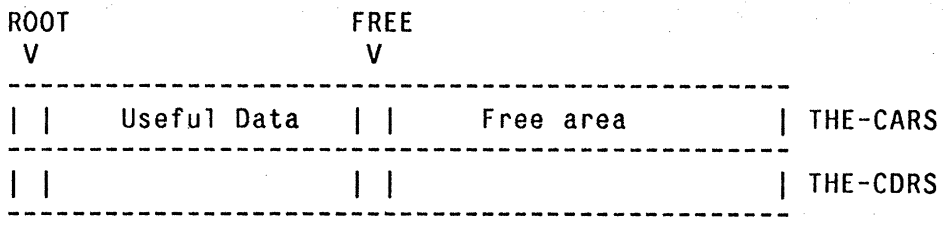
[7] Survey article...

can then be reused next time to copy into.[8]

Let's look at a stop-and-copy garbage collector algorithm to see how it works. We will assume, for the sake of simplicity, that there is a special register, called *root* which contains a pointer to a structure which eventually points at all accessible data. This can be easily arranged by storing all interpreter registers into a pre-allocated list structure pointed at by *root* just before entering the garbage-collection process. We also assume that, besides the current working memory (represented by the contents of the registers *the-cars* and *the-cdrs*) there is free memory to receive the clean copy of useful memory, (in the contents of the registers *new-cars* and *new-cdrs*). We enter the garbage collector when we have run out of free cells in the current working memory. The result of garbage collection is that the *root* points at cells in the new memory, all things accessible from the root are in new memory, and the *free* pointer now points at the next place in new memory where a *cons* can be made. In addition, the new memory and the current memory are interchanged (flipped) so that next time we can repeat the process:



[8] This idea was invented and first implemented on the RLE PDP-1 LISP by Marvin Minsky. It was further developed by Fenichel and Yochelson to be used in the LISP on Multics. Later, Baker made a real-time version of the algorithm, which does not require the LISP to stop during garbage collection. The Minsky-Fenichel-Yochelson algorithm is the dominant algorithm in use for big machines now, because one need not look at all of memory, only the useful part of memory. The MacLISP on the DEC-20/60 is really a small machine LISP, it uses a mark-sweep algorithm. This is acceptible because a 20/60 can only address 2↑18 words of memory.



Just after garbage collection

In this algorithm, garbage collection proceeds by copying all of the valuable data (reachable from the root) sequentially into a clean part of memory. The clean part of memory is initialized with a *free* pointer and a *scan* pointer pointed at its beginning. The first thing done is the *root* is relocated. This is accomplished as follows: The object pointed at by the root is copied into that place, the free pointer is incremented, and the root is adjusted to point at the new home. In addition, the old object which is copied is adjusted to have a forwarding address pointing at the new home.

After initialization (relocation of the root) the garbage collector enters its basic cycle. The *scan* pointer (initially pointing at the new copy of the object pointed at by the root) points at an object which has been moved, but whose *car* and *cdr* still refer to objects in the current memory. The *car* and the *cdr* of the object pointed at by *scan* are each relocated, and the *scan* pointer is incremented to scan the next new cell. Relocation of an object means copying it from the current memory to a new free place, if it has not been copied already. The cell being scanned is then adjusted to point at the new address in new memory. If the object pointed at by *scan* has been copied already its *car* will contain a *broken-heart* (a unique, recognizable object which may not ever be produced by the interpreter -- it is placed there by the copying process) and its *cdr* will contain a forwarding address to the place where it has been moved in the new memory. If it has been moved already, its forwarding address is substituted for the object in the cell being scanned. Eventually, all of the cells which are accessible will have been moved and scanned, thus the *scan* pointer will point at the next *free* place in new memory, and the process will stop. When this happens, we interchange the roles of new and current memory and continue the interpretation process.

Let's now look at the code in more detail. The only very tricky things are the ways in which relocation happens, and in the manipulation of broken hearts and forwarding pointers. Relocating an object, which may involve copying it from the current memory to the new memory, is done by a subroutine called *relocate-old-result-in-new*. It takes its argument in the *old* register and returns the relocation of its argument in the *new* register. We invoke it by storing a return address in the register *relocate-return* and going to the entry *relocate-old-result-in-new*. The garbage collector initialization does precisely that to get the relocation of the *root* after setting up *free* and *scan*:

```
(assign free 0)
(assign scan 0)
(assign old (fetch root))
(assign relocate-return reassign-root)
(goto relocate-old-result-in-new)
```

When the relocation of *root* has been computed, we reinstall it and then enter the main loop of the garbage collector:

```
reassign-root
  (assign root (fetch new))
  (goto gc-done?)
```

In the main loop of the garbage collector we must determine if there are any more objects whose *car* or *cdr* fields need to be relocated. If there are, then the *scan* pointer must be pointing at one of them. If the *scan* pointer is coincident with the *free* pointer then we are done and must go off to the *gc-flip* routine to clean up for next time. If there is indeed material to be scanned, we should first scan the *car* field of the object. Thus the *old* register is set up with the object to be relocated, and we go off to relocate it, expecting to come back to *update-car*:

```
gc-done?
  (branch (= (fetch scan) (fetch free)) gc-flip)
  (assign old (vector-ref (fetch new-cars) (fetch scan)))
  (assign relocate-return update-car)
  (goto relocate-old-result-in-new)
```

At *update-car* we smash away the correctly relocated value and then set up to work on the *cdr* of the scan pointer. We will return to the *update-cdr* routine when that relocation has been computed:

```
update-car
  (perform (vector-set! (fetch new-cars) (fetch scan) (fetch new)))
  (assign old (vector-ref (fetch new-cdrs) (fetch scan)))
  (assign relocate-return update-cdr)
  (goto relocate-old-result-in-new)
```

After having relocated the *cdr* of the scanned object we are finished with that object and go on to the next one to be scanned:

```
update-cdr
  (perform (vector-set! (fetch new-cdrs) (fetch scan) (fetch new)))
  (assign scan (1+ (fetch scan)))
  (goto gc-done?)
```

The relocations are computed by *relocate-old-result-in-new* as follows. If the object to be relocated (in *old*) is a pointer, then it must get a relocation, otherwise, we just want it to be unchanged in the result. If it is a pointer which is pointing to an object that has already been moved, then the *car* of the old place the object was contains a *broken-heart* indicating that the *cdr* of the old place contains a relocation address which can be used as the relocation. If the pointer in *old* points at an as yet unmoved object, then we move it to the first free cell in new memory (pointed at by *free*, and then put a relocation address and broken heart in the current memory:

```
relocate-old-result-in-new
  (branch (not (pointer? (fetch old))) non-pointer)
  (branch (broken-heart? (vector-ref (fetch the-cars) (fetch old)))
          already-moved)
  (assign new (fetch free))       ;Will need for return
  (assign free (1+ free))         ;Used one cell
  ;; These copy the CAR and CDR of the old cell to new memory
  (perform (vector-set! (fetch new-cars) (fetch new)
                        (vector-ref (fetch the-cars) (fetch old))))
  (perform (vector-set! (fetch new-cdrs) (fetch new)
                        (vector-ref (fetch the-cdrs) (fetch old))))
  ;; These build the forwarding address and broken heart.
  (perform (vector-set! (fetch the-cars) (fetch old) broken-heart))
  (perform (vector-set! (fetch the-cdrs) (fetch old) (fetch new)))
  (goto (fetch relocate-return))
non-pointer
  (assign new (fetch old))
  (goto (fetch relocate-return))
already-moved
  (assign new (vector-ref (fetch the-cdrs) (fetch old)))
  (goto (fetch relocate-return))
```

When we are all done, we interchange the current and new memories, completing the garbage collection process:

```
gc-flip
  (assign temp (fetch the-cdrs))
  (assign the-cdrs (fetch new-cdrs))
  (assign new-cdrs (fetch temp))
  (assign temp (fetch the-cars))
  (assign the-cars (fetch new-cars))
  (assign new-cars (fetch temp))
  (goto gc-finished)
```

Exercise 5-13: What would it take to really integrate our storage allocation and garbage collection scheme into the explicit control evaluator described in this chapter? How should the need for garbage collection be triggered? Where and how in the interpreter code should it be installed? This is a very tricky question.

5.4. Compilation

Almost every computer is a register-transfer machine which is controlled by programs written in a native machine language based on operations on and assignment to registers. There are two common strategies for bridging the gap between such register-transfer languages and expression-and-procedure languages such as Lisp or Pascal. One which we have already examined is interpretation. Now we will examine an alternative strategy, compilation.

An interpreter is an amazing program which configures a machine whose native language is one language to simulate a machine whose native language another one. The program being interpreted (called the source program) is a data structure of the interpreter. The interpreter walks the source program, executing code whose effect is to simulate the intended behavior of the source program. The interpreter implements the means of combination of the interpreted language as a mechanism for dynamically interconnecting the implementations of the primitive operators of the source language as directed by traversal of the data structure representing the program being interpreted. The primitive operators of the source language

are implemented as a library of subroutines in the native language of the given machine.

A compiler is an equally amazing program which translates a source program written in one language into an equivalent program (called the object program) written in another language. The program to be compiled is a data structure of the compiler. The compiler walks the source program, constructing the object program, which if executed will behave as intended by the author of the source program. The object program will refer to the library of procedures which implement the primitive operators of the source language in the object language.

Implementation of a language by interpretation provides a nicer run-time environment for interactive development and debugging of programs than is found in compiled systems. This is because in an interpreter, the source program is available as data to be manipulated by the debugging features. In addition, the whole library is present and new programs can be constructed and added to the system during debugging, to examine run-time data structures which would not be interesting in a debugged program.

Compilation provides a much more efficient implementation of source programs in the object language. This is because a compiler takes advantage of the fact that the program being implemented is constant (not true if debugging!). An interpreter must examine the program each time it executes it, discovering over and over again, for example, that a particular procedure application has only one argument. In addition, it must prepare for the worst, saving its state before executing a subexpression, because the subexpression might, in general, require an arbitrary evaluation, even if in reality evaluation of the subexpression is trivial and needs only change the value of the VAL register, for example. A compiler looking at the source program would compile this code as a special case.

In light of these alternative advantages of compiled and interpreted modes of implementation, many modern systems (such as Lisp) allow compiled code to be used as primitives in an interpreter system. This allows one to compile the parts of one's program which are thought to be debugged, allowing one to gain the speed advantage, while retaining the interpretive mode of execution for those parts of the program which are in the flux of interactive development and debugging.

5.4.1. Our system

We now present a simple SCHEME compiler to show how the register transfer language object program is constructed from the expression language source program. We will use this to help us solidify our notion of stack discipline which is the key by which a register machine can perform the actions of a recursive expression evaluation. We will also interface our compiler to our interpreter to make an integrated interpreter/compiler development system.

Our compiler is a rather simple one. It compiles for a machine with assumed primitive operations such as *car*, *cdr*, *cons*, environment lookup, and construction and other high-level operators precisely because we are at this point interested most in understanding in detail how a stack is used to control recursions, and not how to implement the compound data construction and selection procedures in terms of the more primitive machine memory. We will have the opportunity to augment the compiler in such a way as to improve at least some of these aspects of compilation. But, for the nonce we will ignore many of the finer

points of compilation, such as register allocation, open coding of primitive operators, or fancy optimization. What we will be concerned with is:

1. The precise rules and conditions of stack discipline.
2. The kinds of efficiency advantages that accrue from having a program studied and summarized by our compiler, prior to execution of our summary.

We will see that whereas blind interpretation of a program requires always preparing for any contingency, a compiler can write code that takes advantage of foreknowledge of what will happen next, thus minimizing irrelevant preparation and cleaning up.

Compiled code and the interpreter interface

Our compiler will take source code in expression-oriented Scheme and produce object code in the register-transfer language that our interpreter is written in. To be compatible with interpreted code, the compiler will obey all of the conventions of register usage that the interpreter is based on. The compiled code will keep any procedures to be called in *fun*, and the arguments to such procedures will be *cons*ed up in *argl*. In fact, the compiler will produce object code which does what the interpreter would do in executing the same source code. This makes it easy to build an integrated system of interpretation and compilation. Though there are more powerful compilation strategies which yield better object code they will not be considered here. The main point of this section is to illustrate the compilation process in a simplified, but still interesting, context.

To make it possible for compiled code to be executed by the interpreter system, it is necessary to slightly modify the interpreter to accomodate the compiled code. The only modification necessary is in *apply-dispatch* where we must distinguish three cases of procedures -- system primitives, compound (interpretive) procedures, and compiled procedures. In the case of a compiled procedure, the interpreter just directly transfers control to the compiled code body:

```
apply-dispatch
  (branch (primitive-procedure? (fetch fun)) primitive-apply)
  (branch (compiled-procedure? (fetch fun)) compiled-apply)
  (branch (compound-procedure? (fetch fun)) compound-apply)
  (goto unknown-procedure-type-error)

compiled-apply
  (assign val (code-of-compiled-closure (fetch fun)))
  (goto (fetch val))
```

Our compiler puts out its object code in the form of a list structure. Part of the output is the list of instructions which comprises the compiled code. This list of instructions contains *label*s which mark places in the list as the beginnings of sequences of instructions to which the compiler has constructed transfers of control (*goto* instructions). It is expected that there is a "loader" program which actually defines the labeled entry points (performing what is essentially the action of a *defentry*). We will not go into any detail about this process.

Compilation strategy

Structurally and functionally a compiler is like an interpreter. It walks the expression tree dispatching on the type of each expression to a code-generator procedure for that kind of

expression. The code generators "meta-evaluate" each kind of expression in that, rather than executing the expression at the moment, they construct code which has the effect of executing the expression when it is run.

After having examined an expression, a compiler knows a great deal about it. It knows what registers were actually modified in executing that piece of code. For example, no registers except *val* are actually modified in executing a quoted expression or in looking up a value of a variable in the environment. Thus such primitive operations cannot destroy the value of the *env* register, so it need not be *saved* before executing such an expression as a subexpression, to be *restored* after the subexpression execution is finished, even if the value of the *env* register will be indeed needed then.

Each code generator returns a *sequence of instructions* as its value. An instruction sequence contains three pieces of information: The first is the list of *statements* which implements the execution of the expression which was compiled by that code generator. These are the *statements* of the sequence. Secondly, we need the set of registers whose values are modified by the sequence of instructions. This is called the *mung-list* of the sequence.[9] Finally, we need the set of registers which much be set up with values before entering this sequence. This is called the *needs-list*.

A sequence of instructions is constructed as either a primitive instruction (by a constructor like *make-register-assignment*) or by appending instruction sequences using *append-instruction-sequences*. This produces the instruction sequence which is formed from the concatenation of the statement lists of the two sequences. The resulting sequence (potentially) modifies the registers (potentially) modified by either subsequence. It also needs the set of registers needed by the first sequence and those registers needed by the second sequence which were not set up by the first. *Append-instruction-sequences* is built using *Make-seq*. *Make-seq* is the constructor for an instruction sequence with a given *needs-list*, a given *mung-list*, and a given list of *statements*.

```
(define (append-instruction-sequences s1 s2)
  (make-seq (set-union (needs-list s1)
                       (set-difference (needs-list s2)
                                       (mung-list s1)))
            (set-union (mung-list s1) (mung-list s2))
            (append (statements s1) (statements s2))))
```

In addition to the expression, *exp*, to be compiled, each code generator takes several additional arguments. There is a compile-time environment, *env*, which carries a description of the environment which will be in effect when the expression is run. This is only used in sophisticated compilation of variable references -- we will develop this in section 5.4.5. The compile-time environment is extended in the compilation of *lambda* expressions and it is modified by the compilation of definitions. Another argument supplied to each code generator is the *target*. It specifies the register which the compiler wants to use to get the value of the expression. Finally, each code generator takes a *continuation* descriptor, which describes how the code resulting from the compilation of the current expression should proceed when it has finished its execution. The continuation descriptor can require that the code:

---

[9] "Mung" is a recursive acronym standing for "Mung Until No Good". For more bad self-referential jokes like that see Doug Hofstader.

1. *Return* to the caller.
2. Continue at the *next* instruction in sequence.
3. Continue at a named label.

Given such a continuation descriptor we can construct the correct way to continue at that continuation using the procedure *continue-at*. This piece of code will be called all through the compiler to manufacture appropriate unconditional transfer of control instructions:

```
(define (continue-at continuation)
  (cond ((eq? continuation 'return)
         (append-instruction-sequences
          (make-restore 'continue)
          (make-goto-instruction (make-fetch 'continue))))
        ((eq? continuation 'next)
         (the-empty-instruction-sequence))
        (else
         (make-goto-instruction continuation))))
```

We will first go through the code of the compiler, examining the code generators for each kind of expression. Later we will look at an example of compilation, examining the dynamic behavior of the compiler.

The core of the compiler

The program *compile-expression* is the central element in the compiler, corresponding to the *eval* procedure of the meta-circular evaluator, and the *eval-dispatch* procedure of the explicit control evaluator. It takes four arguments, the *exp*ression to be compiled, a description of the *env*ironment which will be available to the compiled code (containing the variable names which are bound, but not their values), a *target* register which will get the value of the expression, and a *cont*inuation descriptor which describes how the compiled expression must continue when it is done.

Compile-expression is a case analysis on the syntactic type of the expression to be compiled. For each type of expression (special form) it dispatches to a specialized code generator for expressions of that type:

```
(define (compile-expression exp env target cont)
  (cond ((self-evaluating? exp)
         (compile-constant exp target cont))
        ((quoted? exp)
         (compile-constant (text-of-quotation exp) target cont))
        ((variable? exp)
         (compile-variable-access exp env target cont))
        ((assignment? exp)
         (compile-assignment exp env target cont))
        ((definition? exp)
         (compile-definition exp env target cont))
        ((lambda? exp)
         (compile-lambda exp env target cont))
        ((conditional? exp)
         (compile-cond (clauses exp) env target cont))
        ((no-args? exp)
         (compile-no-args exp env target cont))
        ((application? exp)
         (compile-application exp env target cont))
        (else
         (error "Unknown expression type -- COMPILE" exp))))
```

Simple code generators for constants and variables construct simple instruction sequences which smash the *target* register with the value required and then continue at the correct continuation:

```
(define (compile-constant constant target cont)
  (append-instruction-sequences
   (make-register-assignment target (make-constant constant))
   (continue-at cont)))

(define (compile-variable-access var env target cont)
  (append-instruction-sequences
   (make-register-assignment target (make-variable-access var env))
   (continue-at cont)))
```

In these fragments we see some of the basic operations of the compiler. The code generator constructs an instruction sequence by appending together two smaller instruction sequences. The first is the sequence of instructions required to assign the value of interest to the *target* register and the second is the sequence of instructions required to continue after the evaluation. An instruction sequence may contain zero or more instructions. If the *cont* is *next*, the *(continue-at cont)* will produce an empty instruction sequence.

Preserving registers

Assignments and definitions are somewhat more complex. Before performing the variable assignment or binding instructions, it is first necessary to compute the value of the expression to which the variable will be bound. We compile the subexpression evaluation with the continuation specifier *next*, indicating that the subexpression evaluation code should continue at the next instruction in the sequence.

The subexpression evaluation must be performed in such a way so that the environment in effect after the evaluation must be the same as the one in effect before the evaluation. Thus the environment register, *env* must, in general, be *saved* and *restored* around the subexpression evaluation. This is accomplished by wrapping the subexpression compilation with a *preserving* form.

Assignments and definitions are usually done where the value is ignored (and only the effect matters). Thus, in these cases, we may specify a target of *nil* (for example, we will see this targeting in the compilation of the elements of a sequence. In that case, the target is taken to be *val*. Alternative strategies may lead to better results:

```
(define (compile-assignment exp env target cont)
  (let ((target (if (null? target) 'val target)))
    (preserving 'env
      (compile-expression (assignment-value exp) env target 'next)
      (append-instruction-sequences
       (make-variable-assignment (assignment-variable exp)
                                 env
                                 (make-fetch target))
       (continue-at cont)))))
```

Assignments change the values of variables already existing in the environment of a running program. Definitions differ from assignments in that the compilation environment used to compile the body of a definition must be augmented to include the variable being defined. In addition, this augmentation must be visible when we compile code which appears later in a sequence of expressions. Thus the compiler environment must be *modified* by the definition. (Note the "!" which conventionally indicates a side-effect operation.)

```
(define (compile-definition exp env target cont)
  (let ((target (if (null? target) 'val target)))
    (preserving 'env
     (compile-expression (definition-value exp)
                         (definition-env! (definition-variable exp)
                           env)
                         target
                         'next)
     (append-instruction-sequences
      (make-variable-definition (definition-variable exp)
                                env
                                (make-fetch target))
      (continue-at cont)))))
```

The procedure *preserving* appends two instruction sequences so that the register named as its first argument is preserved over the execution of the compiled code which is its second argument *if it is needed* in the execution of the compiled code which is its third argument. The general idea is that it wraps a *save* and *restore* of the named register around the first sequence, *if necessary*. There are two ways which the *save* and *restore* may not be necessary -- either the first code sequence does not, in fact, clobber the register being preserved, or the second code sequence really does not need it.

In case first sequence cannot clobber the register preserved or the code after the protected evaluation (the second sequence in the *preserving*) does not actually need the value of the protected register there is no reason to put out the *save* and *restore*. *Preserving* can check for this case by looking at the instruction sequence produced by the compilation.[10] This kind of "push-pop" optimization is available in compiled code because a compiler can study a program, discovering the special properties it has.

In general, however, *preserving* wraps a *save* and *restore* around the compilation of its second argument. *Preserving* thus first checks a sequence of instructions to see if the register it must preserve is in danger of being clobbered by that sequence of instructions. It does this by looking in the *mung-list*. If the register is not in danger, *preserving* returns the original instruction sequence; if it is in danger, *preserving* wraps it up in a *save* -- *restore* pair, and returns the new instruction sequence with the preserved register removed from its *mung-list*.

```
(define (preserving reg seq1 seq2)
  (if (and (memq reg (needs-list seq2))
           (memq reg (mung-list seq1)))
      (append-instruction-sequences
       (make-seq (needs-list seq1)
                 (set-difference (mung-list seq1) (list reg))
                 (append (statements (make-save reg))
                         (statements seq1)
                         (statements (make-restore reg))))
       seq2)
      (append-instruction-sequences seq1 seq2)))
```

The compilation of *sequence*s is quite parallel to the evaluation of them. Each expression

---

[10] In fact, an instruction sequence has both the instructions and a set of registers clobbered by those instructions. A register can only be clobbered by an *assign* instruction which is not wrapped in a *save* -- *restore* pair. Calls to procedures are assumed to clobber all registers. A more sophisticated compiler would know the properties of important primitives, such as *1+*.

of the sequence, except for the last one is compiled preserving the environment. The last expression is compiled without worrying about the environment because presumably, if it were needed, the compilation of the continuation would have compiled the sequence preserving the environment. In addition, non-final expressions are compiled with continuation *next* and target *nil* (which means we don't care) whereas the final expression is compiled with target the target of the sequence and with continuation the continuation of the sequence. The instruction sequences of these are appended together to make one long one. Here we see another kind of optimization performed by compiling a program. An interpreter must continually check if an expression is the last one in a sequence. This must be done each time we go to the next expression in the sequence and each time we encounter the same sequence. The compiler needs to do this only once for each expression when compiling the sequence. It builds code which "knows" where in the sequence it is, thus not needing explicit run-time tests.

```
(define (compile-sequence seq env target cont)
  (if (last-exp? seq)
      (compile-expression (first-exp seq) env target cont)
      (preserving 'env
                  (compile-expression (first-exp seq) env 'nil 'next)
                  (compile-sequence (rest-exps seq) env target cont))))
```

Combinations

The essence of the compilation process is brought out most effectively in the context of the compilation of combinations. What has to be done here is that an instruction sequence has to be formed which first computes the procedure (from the operator expression) and puts it into *fun*. It then computes the arguments (from the operand expressions) and *cons*es them up into *argl*. Finally it must transfer control to the procedure in *fun* with the arguments in *argl*. The procedure must return to the continuation of the calling combination. The *env* and *argl* must be saved and restored as needed in this process. The summary is clear:

```
(define (compile-application app env target cont)
  (preserving 'env
              (compile-expression (operator app) env 'fun 'next)
              (preserving 'fun
                          (compile-operands (operands app) env)
                          (make-call target cont))))
```

The operator was easily computed. We targeted it to *fun* and continued at *next*. We preserved *env* so we can use it to compute the arguments from the operand expressions.

The operands are a bit more tricky. There are three classes of operands. The first argument is computed without an *argl* allocated yet. Thus, the first argument must return to construct the initial *argl*. A normal next argument already has an *argl* prepared to hold it. It must be computed preserving that *argl* and *cons*ing itself onto that *argl* when it is finished. A final argument need not save the environment across its evaluation because the application of the procedure will not need the environment. *Compile-operands* must compile the first operand. If that is the only operand (perhaps there is only one operand) then the resulting instruction sequence is returned. If not, then we form an instruction sequence by appending the sequence of instructions resulting from adding operations to preserve the environment around the compilation of the first operand to the sequence of instructions resulting from compiling the rest of the operands:

```
(define (compile-operands rands env)
  (let ((fo (compile-first-operand rands env)))
    (if (last-operand? rands)
        fo
        (preserving 'env
                    fo
                    (compile-rest-operands (rest-operands rands) env)))))
```

The compilation of the first operand is to the *val* register with the continuation *next*. The *argl* (the list of evaluated arguments) is initialized here to the singleton of the value which appeared in *val* as the value of the first operand:

```
(define (compile-first-operand rands env)
  (append-instruction-sequences
   (compile-expression (first-operand rands) env 'val 'next)
   (make-register-assignment 'argl
                             (make-singleton-arglist (make-fetch 'val)))))
```

Each additional operand is compiled in succession, using *compile-next-operand*. In each case, *env* is only preserved around the evaluation if there are more operands to go (which may need to get at the environment):

```
(define (compile-rest-operands rands env)
  (let ((no (compile-next-operand rands env)))
    (if (last-operand? rands)
        no
        (preserving 'env
                    no
                    (compile-rest-operands (rest-operands rands) env)))))
```

We compile each operand (other than the first) *preserving* the *argl*, because we must *cons* up our value (in *val*) into it and because it will be used in the call of the procedure later:

```
(define (compile-next-operand rands env)
  (preserving 'argl
              (compile-expression (first-operand rands) env 'val 'next)
              (make-register-assignment 'argl
                 (make-addition-to-arglist (make-fetch 'val)
                                           (make-fetch 'argl)))))
```

Combinations with no operands are treated specially, as they are in the interpreter. In fact, all that is needed is to compile the operator targeting the value to *fun*, initializing the *argl* to the empty argument list, and then performing a call of the procedure:

```
(define (compile-no-args app env target cont)
  (append-instruction-sequences
   (compile-expression (operator app) env 'fun 'next)
   (append-instruction-sequences
    (make-register-assignment 'argl (make-empty-arglist))
    (make-call target cont))))
```

Finally, the compilation of each combination must finish with a call to the procedure in *fun*. The form of this procedure call depends upon the target of the procedure value and upon the type of continuation. Procedures are expected, in this system, to return their values in the *val* register. Thus, if we want the value of a procedure in a different register (say the *fun* register, for example) we must output a special assignment operation to assign to the desired target register the contents of the *val* register after the procedure returns:

```
(define (make-call target cont)
  (let ((cc (make-call-result-in-val cont)))
    (if (eq? target 'val)
        cc
        (append-instruction-sequences
         cc
         (make-register-assignment target (make-fetch 'val))))))
```

The basic procedure call (which expects the result in *val*) is compiled differently depending upon the kind of continuation. If the caller will just return with the answer (*cont* = *return*) then we want to transfer to the procedure in *fun*, allowing it to return to the real caller. This ensures tail recursive implementation. If the caller has more to do after the current procedure call, then it will either supply the statement label to continue at or the default label *next*. This must be set up as a saved continuation on the stack. If the continuation is *next*, a label is constructed and set up as the beginning of the sequence after the transfer to the procedure applicator. In this system the procedure applicator is *apply-dispatch*. Thus this is an abstract transfer to the interpreter entry for applying a procedure which is assumed to be in the *fun* register to arguments which are assumed to be in the *argl* register.

```
(define (make-call-result-in-val cont)
  (cond ((eq? cont 'return)
         (make-transfer-to-procedure-applicator))
        ((eq? cont 'next)
         (let ((after-call (generate-new-name 'after-call)))
           (append-instruction-sequences
            (make-call-return-to after-call)
            (make-label after-call))))
        (else
         (make-call-return-to cont))))                    ;A label
```

If the calling program expects to continue after the procedure being called is done, the return label of the continuation must be saved on the stack because the application is being done by *apply-dispatch* which assumes that the continuation is *saved* and will be restored when it will be needed.

```
(define (make-call-return-to retlabel)
  (append-instruction-sequences
   (append-instruction-sequences
    (make-register-assignment 'continue retlabel)
    (make-save 'continue))
   (make-transfer-to-procedure-applicator)))
```

Conditional expressions

Compilation of conditional expressions does a considerable amount of manipulation of labels to organize the flow of control. The code-generator for conditionals first determines if the conditional expression is expected to return in line. If so, it must make up a label to continue at, so that the various branches have a way of rejoining. If the conditional is supplied with a continuation label, the branches will be compiled with that label for rejoining. If it is a *return* continuation, each branch will be expected to return when it runs out:

```
(define (compile-cond clauses env target cont)
  (if (eq? cont 'next)
      (let ((end-of-cond (generate-new-name 'cond-end)))
        (append-instruction-sequences
         (compile-clauses clauses env target end-of-cond)
         (make-label end-of-cond)))     ;Output label
      (compile-clauses clauses env target cont)))
```

The compilation of the clauses is quite straightfoward. In the general case, if there are clauses to compile and the current clause is not an *else* clause, then the output instruction sequence is just a compilation of the predicate expression to *val*, followed by a test of the *val* register which has two alternatives, the compiled consequent and the compilation of the rest of the clauses of the *if*. The compilation of the predicate continues at the branch (*next*). The branches of the test compile to the target of the conditional and continue at the continuation of the conditional. Note how it is important to preserve *env* around the evaluation of the predicate -- we will need it for the consequent and the alternative.

```
(define (compile-clauses clauses env target cont)
  (if (no-clauses? clauses)
      (continue-at cont)
      (let ((fc (first-clause clauses)))
        (if (else-clause? fc)
            (compile-sequence (action-sequence fc) env target cont)
            (let ((ift (generate-new-name 'true-branch)))
              (preserving 'env
               (compile-expression (predicate fc) env 'val 'next)
               (append-instruction-sequences
                (make-branch (make-fetch 'val) ift)
                (join-instruction-sequences
                 (compile-clauses (rest-clauses clauses)
                                  env target cont)
                 (append-instruction-sequences
                  (make-labeled-point ift)
                  (compile-sequence (action-sequence fc)
                                    env target cont)))))))))))
```

We see that here we want to make an instruction sequence which is not intended to be executed sequentially. There are two branches to the conditional, only one of which may be traversed in any particular execution. Thus, these must be appended differently than the normal sequential instruction-sequence append:

```
(define (join-instruction-sequences s1 s2)
  (make-seq (set-union (needs-list s1) (needs-list s2))
            (set-union (mung-list s1) (mung-list s2))
            (append (statements s1) (statements s2))))
```

Compiling LAMBDA expressions

Lambda expressions are the only special forms we have not yet considered. A *lambda* expression is a constructor for procedures. A procedure needs to know two pieces of information, the *script* it must follow, and the *environment* for interpretation of its free variables. The script can be represented by the label marking the entry point to the compiled body of the *lambda* expression. The only interesting point here is what must happen if the continuation is *next*. In that case the code sequence, which contains the compilation of the *lambda* body must jump over the compiled body:

```
(define (compile-lambda exp env target cont)
  (let ((entry (generate-new-name 'entry)))
    (append-instruction-sequences
     (make-register-assignment target (make-procedure-maker entry))
     (if (eq? cont 'next)
         (let ((after-lambda (generate-new-name 'after-lambda)))
           (append-instruction-sequences
            (continue-at after-lambda)
            (append-instruction-sequences
             (compile-lambda-body exp env entry)
             (make-label after-lambda))))
         (append-instruction-sequences
          (continue-at cont)
          (compile-lambda-body exp env entry))))))
```

The script of a *lambda* expression is also straightforward. It begins with a label for the entry point. The procedure then proceeds to switch environments to the environment of the definition of the *lambda* expression (the lexical context) extended so as to include the arguments mapped to the formal parameters. The rest is the compilation of the body of the *lambda* expression in an environment where the formal parameters of the expression are bound. The procedure is presumed to return when its body is completed.

```
(define (compile-lambda-body exp env entry)
  (safe-instruction-sequence
   (append-instruction-sequences
    (make-label entry)
    (append-instruction-sequences
     (make-environment-switch (lambda-parameters exp))
     (compile-sequence (lambda-body exp)
                       (extend-compile-time-env (lambda-parameters exp) env)
                       'val
                       'return)))))
```

The only funny thing here is the procedure *safe-instruction-sequence*. This hides the registers clobbered by the compiled body of the *lambda* expression from the sequence in which it is embedded. This is reasonable because the body of the *lambda* expression is not "in line" to be executed as part of the sequence. In fact the only way it can be entered is from a procedure call, which is in general assumed to clobber all registers. Thus *safe-instruction-sequence* constructs a sequence of instructions with a null *mung-list* and the list of statements from the given sequence.

5.4.2. Representations

The code we have seen so far is pretty abstract. We have used no knowledge of the format of an object code instruction. Instead, it concentrates on the ways by which instructions are combined to make coherent programs. Eventually, however, we have to get down to the details. The instructions we use are from the abstract machine language we defined our explicit control evaluator in. It contains *goto*s, *assign*ment to registers, *fetch*ing of registers, and a fixed number of primitive operators. For example, to *continue-at* a particular continuation, we must produce an instruction sequence which contains an appropriate *goto* operation. The procedure *make-instruction* takes a "machine language" instruction and makes a sequence with it as the only statement. The first argument specifies the set of registers clobbered by that instruction. This is how the set of registers clobbered are ultimately specified. *Goto* instructions clobber no registers, but they may need

the value of a register, such as *continue*, to use as a label.

```
(define (make-goto-instruction continuation)
  (make-instruction (needs-list continuation)
                    '()
                    (list 'goto (value-of continuation))))
```

In our abstract machine language a conditional test and branch is represented as follows:

```
(define (make-branch predicate if-true-label)
  (make-instruction (needs-list predicate)
                    '()
                    (list 'branch
                          (value-of predicate)
                          if-true-label)))
```

Indeed, it is necessary for the compiler to know some properties of the interpreter system. We will later see a few minor modifications which must be made to the interpreter to accomodate compiled code. In the following code we see that all of the registers are potentially modified by calling a procedure. *All* is a list of all of the registers. Notice that procedure application only needs *fun* and *argl* to work.

```
(define (make-transfer-to-procedure-applicator)
  (make-instruction '(fun argl) all '(goto apply-dispatch)))
```

Labels are simply notated. They are not used in the interpreter. We will have to make a *loader* which inserts compiled code into the interpreter system. It will change a label into an *entry* point

```
(define (make-labeled-point label)
  (make-instruction '() '() label))
```

Register assignments are the most common "machine language" statements. If the target register of an assignment is *nil* we assume that there is really no need to do the assignment:

```
(define (make-register-assignment reg val)
  (cond ((not (null? reg))
         (make-instruction (needs-list val)
                           (list reg)
                           (list 'assign reg (value-of val))))
        (else
         (the-empty-instruction-sequence))))
```

Besides assignment, the standard register operations are needed in the compiler. *Fetch* is a value fragment, such as an indicated constant. It has two parts. The actual value expression (which is extracted by *value-of*) and the needs list (which is extracted by *needs-of*). The needs list of a register fetch is a singleton containing just that register:

```
(define (make-fetch reg)
  (make-value (list reg) (list 'fetch reg)))
```

The stack operations are also simple instructions. A *save* has no needs and changes no registers.[11]

```
(define (make-save reg)
  (make-instruction '() '() (list 'save reg)))
```

---

[11] Actually, this is a rather unsophisticated compiler. In a more clever design, a *save* would be thought of as needing the value of the register being *saved* and changing the value of *the-stack*. This would lead to a more coherent model of the computation which could be used to make more optimal code

In this naive model, *restore* is very similar to *save*.

```
(define (make-restore reg)
  (make-instruction '() '() (list 'restore reg)))
```

In the previous operations, *fetch* is different from the others in that it is not a complete instruction, but rather it is a value fragment which is used to specify a source of data for an instruction. We need other such fragment constructors as well:

```
(define (make-constant x)
  (make-value '() (list 'quote x)))
(define (make-variable-access var compilation-env)
  (make-value '(env)
              (list 'lookup-variable-value
                    (list 'quote var)
                    (value-of (make-fetch 'env)))))
```

Variable assignments and definitions turn into simple instructions which are probably abstractions of rather complex processes in real machines. We use the *perform* syntax to indicate such instructions which are executed for effect on data structures or input-output devices.

```
(define (make-variable-assignment var compilation-env val)
  (make-instruction (set-union '(env) (needs-list val))
                    '()
                    (list 'perform
                          (list 'set-variable-value!
                                (list 'quote var)
                                (value-of val)
                                (value-of (make-fetch 'env))))))
(define (make-variable-definition var compilation-env val)
  (make-instruction (set-union '(env) (needs-list val))
                    '()
                    (list 'perform
                          (list 'define-variable!
                                (list 'quote var)
                                (value-of val)
                                (value-of (make-fetch 'env))))))
```

There are a few other simple instruction fragments we need. Environments are extended with the interpreter routine *extend-environment*.

```
(define (make-bindings-maker vars args env)
  (make-value (list (needs-list args) (needs-list env))
              (list 'extend-environment
                    (list 'quote vars)
                    (value-of args)
                    (value-of env))))
```

Lambda expressions compile into code for the construction of procedures. Procedure objects are constructed from compiled code by a special routine, *make-compiled-closure*, which combines a compiled procedure's entry point with the current environment.

```
(define (make-procedure-maker entry)
  (make-value '(env)
              (list 'make-compiled-closure
                    entry
                    (value-of (make-fetch 'env)))))
```

When a compiled procedure is entered, it is necessary to switch to the environment of

definition of the procedure-defining *lambda* expression extended by the binding of the procedure's formal parameters to its arguments. This requires that procedure objects be packaged up with their environment, as above. The environment is extracted from a compiled procedure object with a special selector procedure, *env-of-compiled-closure*:

```
(define (make-env-ref fun)
  (make-value (needs-list fun)
              (list 'env-of-compiled-closure
                    (value-of fun))))
```

As in the interpreter, the list of evaluated arguments is started at *nil* and is *cons*ed up from pairs:

```
(define (make-singleton-arglist val)
  (make-value (needs-list val)
              (list 'cons (value-of val) '())))

(define (make-addition-to-arglist val args)
  (make-value (set-union (needs-list val) (needs-list args))
              (list 'cons (value-of val) (value-of args))))
```

There are just a few, low level details that have to be attended to in this compiler. For example, instructions and instruction sequences must be represented:

```
(define (make-value needed-regs expression)
  (list needed-regs expression))

(define (needs-list value)
  (if (symbol? value)                    ; Label
      '()
      (car value)))

(define (value-of value)
  (if (symbol? value)
      value
      (cadr value)))

(define (make-instruction needs mungs code)
  (make-seq needs mungs (list code)))

(define (make-seq needs mungs seq)
  (list needs mungs seq))

(define (the-empty-instruction-sequence)
  (list '() '() '()))

(define (mung-list seq) (cadr seq))
(define (statements seq) (caddr seq))

(define (safe-instruction-sequence seq)
  (make-seq '() (statements seq)))
```

5.4.3. An example of compilation

To help us understand how our compiler actually works, it is useful to follow out the beginning of the compilation of our simple recursive factorial program:

```
(define (rfact n)
  (cond ((< n 2) 1)
        (else (* n (rfact (- n 1))))))
```

We start up by calling the *compile* procedure with the text of the definition as the expression. *compile* just calls *compile-expression* (see page 342) with the given expression, with the null environment, targeting the result to *val*, and continuing by returning

the value developed:

```
(define (compile exp)
  (compile-expression exp '() 'val 'return))
```

compile-expression is the compilation analog of *eval* in an interpreter. It dispatches on the expression type to the various code generators. We see that simple expressions like self-evaluating, quoted expressions, and variables, are compiled on the spot. They just compile into a sequence of a register assignment and the continuation which was specified for the expression. The assignment is to the register which was specified in the *target* argument. Since we were compiling a definition, the compiler dispatches to the specific code generator *compile-definition* (see page 344) for definitions.

The code compiled is then constructed as a sequence of: The code required to evaluate the definiens, the code required to modify the environment to install the definition, and the continuation for after executing the definition (In this case the continuation is *return*, because we are presuming a top-level definition.). Notice how the continuation of the definiens is continued at the *next*. In addition, the computation of the value of the definiens is done *preserving* the *env* register's value. This is because we will need the environment of the definition to install the newly defined variable with its value. One final point is that the compilation of the value of the definiens is done in the environment assuming that the definition has been done. Thus the code of the definiens can refer to the variable defined. The compiler tries to maintain an environment parallel to the one to be available at run time. The current compiler doesn't use its environment -- for example, for compiling variable references -- it compiles variable references to go through the interpreter's search mechanism. We will fix this in the last set of exercises. Thus the compilation of the definition of factorial yields the following skeleton:

```
((VAL)
 (<compilation of the definiens, preserving ENV, target VAL>
  (DEFINE-VARIABLE! 'RFACT (FETCH VAL) (FETCH ENV))
  (GOTO RETURN)))
```

Now the definiens of the definition we are compiling is

```
(lambda (n)
  (cond ((= n 0) 1)
        (else (* n (rfact (- n 1))))))
```

so *compile-expression* dispatches to *compile-lambda* (see page 349) where the procedure definition is further compiled. Here, code is produced to construct the assignment to the target register of the closure of the compiled procedure with its environment. The sequence must then jump around the procedure body (which is given an entry-point and compiled in-line) or it must continue at the given continuation if it is not just part of an in-line sequence.

This compilation of the definiens produces the following refinement of the skeleton. In the current case, the continuation of the definiens is *next* so the code jumps around the body.

```
((ENV CONTINUE)    (VAL)
 ((ASSIGN VAL (MAKE-COMPILED-CLOSURE ENTRY1 (FETCH ENV)))
  (GOTO AFTER-LAMBDA2)
  ENTRY1
  <compilation of the definiens>
  AFTER-LAMBDA2
  (PERFORM (DEFINE-VARIABLE! 'RFACT (FETCH VAL) (FETCH ENV)))
  (RESTORE CONTINUE)
  (GOTO (FETCH CONTINUE))))
```

Now the body of the lambda expression must be compiled. It must have the given entry point, *entry1*, and it must bind the arguments to the formal parameters. In addition, the presence of the lambda body in the sequence of instructions does not destroy any registers, because that code is not executed by the sequence, only by the call. This code generator produces the following instructions in this case:

```
((ENV CONTINUE)    (VAL)
 ((ASSIGN VAL (MAKE-COMPILED-CLOSURE ENTRY1 (FETCH ENV)))
  (GOTO AFTER-LAMBDA2)
  ENTRY1
  (ASSIGN ENV (ENV-OF-COMPILED-CLOSURE (FETCH FUN)))
  (ASSIGN ENV (EXTEND-ENVIRONMENT '(N) (FETCH ARGL) (FETCH ENV)))
  <compilation of the body of the definiens>
  AFTER-LAMBDA2
  (PERFORM (DEFINE-VARIABLE! 'RFACT (FETCH VAL) (FETCH ENV)))
  (RESTORE CONTINUE)
  (GOTO (FETCH CONTINUE))))
```

Ad nauseum. I will not go through any more of this code explicitly. The full compilation of the procedure is as follows:

```
((ENV CONTINUE)    (VAL)
 ((ASSIGN VAL (MAKE-COMPILED-CLOSURE ENTRY1 (FETCH ENV)))
  (GOTO AFTER-LAMBDA2)
  ENTRY1
  (ASSIGN ENV (ENV-OF-COMPILED-CLOSURE (FETCH FUN)))
  (ASSIGN ENV (EXTEND-ENVIRONMENT '(N) (FETCH ARGL) (FETCH ENV)))
  (SAVE ENV)
  (ASSIGN FUN (LOOKUP-VARIABLE-VALUE '< (FETCH ENV)))
  (ASSIGN VAL (LOOKUP-VARIABLE-VALUE 'N (FETCH ENV)))
  (ASSIGN ARGL (CONS (FETCH VAL) NIL))
  (ASSIGN VAL '2)
  (ASSIGN ARGL (CONS (FETCH VAL) (FETCH ARGL)))
  (ASSIGN CONTINUE AFTER-CALL6)
  (SAVE CONTINUE)
  (GOTO APPLY-DISPATCH)
  AFTER-CALL6
  (RESTORE ENV)
  (BRANCH (FETCH VAL) TRUE-BRANCH3)
  (ASSIGN FUN (LOOKUP-VARIABLE-VALUE '* (FETCH ENV)))
  (SAVE FUN)
  (ASSIGN VAL (LOOKUP-VARIABLE-VALUE 'N (FETCH ENV)))
  (ASSIGN ARGL (CONS (FETCH VAL) NIL))
  (SAVE ARGL)
  (ASSIGN FUN (LOOKUP-VARIABLE-VALUE 'RFACT (FETCH ENV)))
  (SAVE FUN)
  (ASSIGN FUN (LOOKUP-VARIABLE-VALUE '- (FETCH ENV)))
  (ASSIGN VAL (LOOKUP-VARIABLE-VALUE 'N (FETCH ENV)))
  (ASSIGN ARGL (CONS (FETCH VAL) NIL))
  (ASSIGN VAL '1)
  (ASSIGN ARGL (CONS (FETCH VAL) (FETCH ARGL)))
  (ASSIGN CONTINUE AFTER-CALL5)
  (SAVE CONTINUE)
  (GOTO APPLY-DISPATCH)
  AFTER-CALL5
  (ASSIGN ARGL (CONS (FETCH VAL) NIL))
  (RESTORE FUN)
  (ASSIGN CONTINUE AFTER-CALL4)
  (SAVE CONTINUE)
  (GOTO APPLY-DISPATCH)
  AFTER-CALL4
  (RESTORE ARGL)
  (ASSIGN ARGL (CONS (FETCH VAL) (FETCH ARGL)))
  (RESTORE FUN)
  (GOTO APPLY-DISPATCH)
  TRUE-BRANCH3
  (ASSIGN VAL '1)
  (RESTORE CONTINUE)
  (GOTO (FETCH CONTINUE))
  AFTER-LAMBDA2
  (PERFORM (DEFINE-VARIABLE! 'RFACT (FETCH VAL) (FETCH ENV)))
  (RESTORE CONTINUE)
  (GOTO (FETCH CONTINUE))))
```

5.4.4. Problem section: Compiled Code

This section requires that you have access to files containing the code of the compiler we have developed in the notes.

We can compile expressions by using the Scheme procedure *compile*. Thus we can see what kind of code some expression will turn into by calling *compile* on that expression and

pretty-printing the answer. Thus, for example, the code above was constructed by typing at Scheme, not the embedded model evaluator. We typed:

```
(define foo
  '(define (rfact n)
     (cond ((< n 2) 1)
           (else (* n (rfact (- n 1)))))))

(pp (compile foo))
```

Exercise 5-14: Consider the compilation of the following procedures for computing factorials:

```
(define (rfact n)
  (cond ((< n 2) 1)
        (else (* n (rfact (- n 1))))))

(define (rrfact n)
  (cond ((< n 2) 1)
        (else (* (rrfact (- n 1)) n))))

(define (ifact n)
  (define (iter count answer)
    (cond ((> count n) answer)
          (else (iter (+ count 1) (* count ans)))))
  (iter 2 1))
```

The compilation of program *rfact* is given in the notes above.

1. Compile program *rrfact* and produce a listing of the resulting register-transfer code. Compare the listings of the compilations of *rfact* and *rrfact*. Are there any differences? Explain the differences you find in the compiled code -- why did the compiler make these differences? Which program executes more efficiently?
2. Compile program *ifact*. Annotate the listings showing the essential difference between *ifact* and *rfact* or *rrfact* that makes *ifact* evolve iterative processes while *rfact* and *rrfact* evolve recursive processes.

We have arranged that our evaluator to be able to call code compiled by the compiler. Thus we can compare the performance of programs compiled with the performance of programs interpreted.

Exercise 5-15: We can consider the run-time properties of compiled code as well as its static properties. You are to make another table, similar to the one made for the interpretive versions of the programs. We want expressions in *n*, the input argument to factorial, which give the approximate numbers of pushes and the approximate maximum depth of stack generated by compiled code for *rfact* and *ifact*. You can execute a compiled program by using the procedure *compile-and-go* from Scheme. This should be used to, for example, load a *define* into the model evaluator, and then use it as follows:

```
---> (compile-and-go '(define (square x) (* x x)))
<COMPILED-PROCEDURE>
(TOTAL-PUSHES 1) (MAXIMUM-DEPTH 1)
**==> (square 4)
16
(TOTAL-PUSHES 7) (MAXIMUM-DEPTH 4)
**==>
```

5.4.5. Problem section: Compiled Lexical Lookup

The compiler we prepared does not do anything very special with variables, though one of the most common optimizations made by compilers is the optimization of variable lookup. The problem is that the interpreter lookup routines search for the variable to be accessed, by comparing each variable in the environment with the one to be looked up. Consider, for example, looking for the value of *x* in an environment which has several frames, some of which have several variables:

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) (* X y z))
      (* a b c y)
      (+ c d e))))
 3
 4)
```

In the interpreter, it is necessary for the lookup routine to search for *x*, noticing that it is not *y* or *z* or *a* or *b* or *c* or *d* or *e*. But the compiler can know that and just realize that *x* is two frames out of the current environment and the first variable in that frame. In the last exercise before we started on the compiler you should have observed that the environments always have a structure which is parallel to the lexical structure of the language. That is how the static scoping works. Here, we will augment the compiler/interpreter system to implement "lexical addressing" in compiled code.

The idea is that we will invent a new kind of lookup operation on environments, *lexical-address-lookup*, which will take two numbers, a frame number, telling how many frames to pass over, and a displacement number, telling which variable to access in that frame, and an environment. *lexical-address-lookup* will then produce the value of the variable stored at that lexical address relative to the current environment. The lexical address of a variable is not constant, but depends upon where we are in the code. For example, in the following program, in expression *E1* the address of *x* 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* the address of *x* is now <1 0>, *y* is <1 1>, and *c* is <1 2>.

```
((lambda (x y)
   (lambda (a b c d e)
     ((lambda (y z) E1)
      E2
      (+ c d e))))
 3
 4)
```

Exercise 5-16: Write *lexical-address-lookup*, taking three arguments, two numbers and an environment, which returns the value of the variable stored at the lexical address given.

Our next problem is getting the compiler to spit out the correct calls to *lexical-address-lookup* rather than *lookup-variable-value*. The basic idea is that the compiler must determine, in its own environment, which is hopefully parallel to that which will be in effect at interpretation time, the position of the desired variable. If the variable does not occur in the compiler's environment it is assumed to be global and must be searched using the interpreter's mechanism, *lookup-variable-value*. So we need a program

which will scan the compiler's environment, which is just a list of the lists of bound variables available, to produce a lexical address -- call it *find-variable*.

Exercise 5-17: For example, in the code above, at position *e1* the compiler will have the compiler environment *((y z) (a b c d e) (x y))*. Write *find-variable* to produce:

```
(find-variable 'c '((y z) (a b c d e) (x y))) --> (1 2)
(find-variable 'x '((y z) (a b c d e) (x y))) --> (2 0)
(find-variable '+ '((y z) (a b c d e) (x y))) --> NIL
```

Exercise 5-18: Now we have enough stuff to do the job. Modify *make-variable-access* to use *find-variable* and output a lexical address lookup in cases where *find-variable* found the variable in question. Install your patches into the compiler and compile *ifact*. List the resulting code and run it to prove that it works.

5.5. Summary

We have learned a bit about the technology of computer language implementation. We have seen a language exposed in terms of a meta-circular *eval-apply* interpreter. Such a formulation is excellent for quickly bringing up and playing with a proposed linguistic idea. It is an excellent medium for discussions of comparative linguistics -- for arguing about features and bugs. For example, when we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community we usually mail him a meta-circular interpreter implementing the change. The recipient can thus play with the new interpreter with his machine, and then he can send back his comments as further modifications.

We have seen a LISP interpreter implemented in a simulated register-transfer language similar to the native machine language of most computers. In this form we saw the control of the interpreter quite clearly. We learned about using a stack to implement recursive procedures. We learned about tail-recursion and iteration. The register-transfer level is an excellent medium for discussing implementation issues, like the ideas of having several stacks or the ideas of having special argument-passing registers. The only non-physical assumptions in this model appear in the storage allocation primitives of the underlying language. We at first assumed the existence of *cons*, *car*, and *cdr* as hardware primitives.

We then examined garbage-collection, a technique designed to maintain the illusion of an infinite, list-structured memory in the face of a finite, linearly-organized memory. We studied a particular storage-allocation and garbage-collection scheme compatable with a register-transfer implementation.

Finally, we studied a simple compiler for code from surface Scheme to the register-transfer form which we used to express our interpreter. We used this to learn about the stack discipline and how it can be optimized. We saw how compiled code can be incorporated into an interpretive run-time environment.

Appendix I
Using Environments to Create Packages

This appendix isn't written yet. See next draft.

References

1. Arvind and J. Dean Brock. Streams and Managers. Proceedings of the 14th IBM Computer Science Symposium, Springer-Verlag Lecture Notes in Computer Science, 1983.

2. John Backus. Can Programming be Liberated from the Von Neumann Style? *Communications of the ACM 21*, 8 (August 1978), 613-641.

3. Alan Borning. ThingLab -- An Object-Oriented System for Building Simulations Using Constraints. Proceedings 5th IJCAI, August, 1977, pp. 497-498.

4. Alan Borodin and Ian Munro. *The Computational Complexity of Algebraic and Numeric Problems.* American Elsevier Publishing Company, New York, N.Y., 1975.

5. Gregory J. Chaitin. Randomness and Mathematical Proof. *Scientific American 232*, 5 (May 1975), 47-52.

6. Alonzo Church. *The Calculi of Lambda-conversion.* Princeton University Press, Princeton, N.J., 1941.

7. William Clinger. Nodeterministic Call by Need is Neither Lazy nor by Name. ACM symposium on LISP and Functional Programming, 1982, pp. 226-234.

8. A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un system de communication homme-machine en francais. Groupe Intelligence Artificielle, Universite d'Aix Marseille, Luminy, 1973.

9. John Darlington, Peter Henderson, and David Turner. *Functional Programming and its Applications.* Cambridge University Press, Cambridge, U.K., 1982.

10. Johan deKleer, John Doyle, Guy Steele, and Gerald J. Sussman. AMORD: Explicit Control of Reasoning. Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages, 1977, pp. 116-125.

11. D.P. Freidman and D.S. Wise. CONS Should not Evaluate its Arguments. Automata, Languages, and Programming: Third International Colloquium, July, 1976, pp. 257-284.

12. Cordell Green and Bertram Raphael. The Use of Theorem-Proving Techniques in Question-Answering Systems. Proc. of ACM National Conference, 1968, pp. 169-181.

13. Cordell Green. Application of Theorem Proving to Problem Solving. Proc. of IJCAI, 1969, pp. 219-240.

14. Martin L. Griss. Portable Standard Lisp, A Brief Overview. Utah Symbolic Computation Group Operating Note 58, University of Utah, December, 1981.

15. John V. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM 20*, 6 (1977").

16. Richard W. Hamming. *Coding and Information Theory.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

17. G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers, Fourth Edition.* Oxford University Press, Oxford, 1960.

18. Anthony C. Hearn. Standard Lisp. Memo AIM-90, Artificial Intelligence Project, Stanford University, March, 1969.

19. Peter Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

20. Carl Eddie Hewitt. PLANNER: A Language for Proving Theorems in Robots. Proc. of IJCAI, 1969, pp. 295-301.

21. Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence 8*, 3 (1977), 323-364.

22. C.A.R. Hoare. Proof of Correctness of Data Representations. *Communications of the ACM 1*, 4 (1972), 271-281.

23. Douglas R. Hofstadter. *Goedel, Escher, Bach: an Eternal Golden Braid.* Basic Books, New York, 1979.

24. Donald E. Knuth. *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Ma., 1969.

25. Robert Kowalski. Predicate Logic as a Programming Language. memo 70, DEpartment of Computational Logic, School of Artificial Intelligence, University of Edinburgh, 1973.

26. Robert Kowalski. *Logic for Problem Solving.* North Holland, New York, 1979.

27. Butler Lampson, J.J. Horning, R. London, J.G. Mitchell, and G.K. Popek. Report on the Programming Language Euclid. Computer Systems Research Group, University of Toronto, 1981.

28. Peter Landin. A Correspondence between Algol 60 and Church's Lambda Notation: Part I. *Comm. ACM 8*, 2 (February 1965), 89-101.

29. Barbara H. Liskov and S. N. Zilles. Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering 1*, 1 (1975).

30. John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Comm. ACM* (April 1960), 184-195.

31. John McCarthy et. al. *Lisp 1.5 Programmer's Manual, 2nd Edition.* MIT Press, Cambridge, Ma., 1965.

32. John McCarthy. The History of Lisp. Proceedings of the ACM SIGPLAN Conference on the History of Programming Languages, ACM, 1978.

33. David Moon. MacLisp Reference Manual, Version 0. MIT Laboratory for Computer Science, April, 1978.

34. David Moon and Daniel Weinreb. Lisp Machine Manual. MIT Artificial Intelligence Laboratory, March, 1981.

35. J.H. Morris, Eric Schmidt, and Philip Wadler. Experience with an applicative string processing language. Proc. 7th Annual ACM SIGACT/SIGPLAN Symposium on POPL, 1980.

36. Ronald Rivest, Adi Shamir and Leonard Adelman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Technical Memo LCS/TM82, MIT Laboratory for Computer Science, April, 1977.

37. J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (January 1965), 23.

38. Robert Solovay and Volker Strassen. A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing* (1977), 84-85.

39. Guy Lewis Steele Jr., and Gerald Jay Sussman. Scheme: An Interpreter for the Extended Lambda Calculus. memo 349, MIT Artificial Intelligence Laboratory, 1975.

40. Guy Lewis Steele Jr. Debunking the 'Expensive Procedure Call' Myth. Proceedings of the National Conference of the ACM, ACM, 1977, pp. 153-62.

41. Guy Lewis Steele Jr. An Overview of Common Lisp. ACM symposium on LISP and Functional Programming, 1982, pp. 98-107.

42. Guy Lewis Steele Jr., *et. al.*. The Hacker's Dictionary. computer file residing on a number of machines in the Arpanet community

43. Joseph E. Stoy. *Denotational Semantics.* MIT Press, Cambridge, Ma., 1977.

44. Gerald J. Sussman, Terry Winograd, and Eugene Charniak. Microplanner Reference Manual. memo 203A, MIT Artificial Intelligence Laboratory, December, 1971.

45. Gerald Jay Sussman and Richard M. Stallman. Heuristic Techniques in Computer-Aided Circuit Analysis. *IEEE Transactions on Circuits and Systems CAS-22*, 11 (November 1975).

46. Gerald Jay Sussman and Guy Lewis Steele Jr. Constraints -- A Language for Expressing Almost-Hierachical Descriptions. *AI Journal 14* (1980), 1-39.

47. Ivan E. Sutherland. SKETCHPAD: A Man-Machine Graphical Communication System. Technical report 296, MIT Lincoln Laboratory, January, 1963.

48. Warren Teitelman. Interlisp Reference Manual. Xerox Palo Alto Research Center, 1974.

49. David Turner. The Future of Applicative Languages. Proceedings of the 3rd European Conference on Informatics, Vol. 123, Springer-Verlag Lecture Notes in Computer Science, October, 1981, pp. 334-348.

50. Richard C. Waters. A Method for Analyzing Loop Programs. *IEEE Trans. on Software Engineering 5*, 3 (May 1979), 237-247.

51. Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical Report AI TR-17, MIT Artificial Intelligence Laboratory, February, 1971.

52. Richard Zippel. Probabilistic Algorithms for Sparse Polynomials. Ph.D. Dissertation, MIT Department of Electrical Engineering and Computer Science

Index

Index complete only through Chapter 3