title page

copyright page

To my family, and to Jackie.

λ

Preface

This book is intended for anyone who wants to become a better Lisp programmer. It assumes some familiarity with Lisp, but not necessarily extensive programming experience. The first few chapters contain a fair amount of review. I hope that these sections will be interesting to more experienced Lisp programmers as well, because they present familiar subjects in a new light.

It's difficult to convey the essence of a programming language in one sentence, but John Foderaro has come close:

Lisp is a programmable programming language.

There is more to Lisp than this, but the ability to bend Lisp to one's will is a large part of what distinguishes a Lisp expert from a novice. As well as writing their programs down toward the language, experienced Lisp programmers build the language up toward their programs. This book teaches how to program in the bottom-up style for which Lisp is inherently well-suited.

Bottom-up Design

Bottom-up design is becoming more important as software grows in complexity. Programs today may have to meet specifications which are extremely complex, or even open-ended. Under such circumstances, the traditional top-down method sometimes breaks down. In its place there has evolved a style of programming quite different from what is currently taught in most computer science courses: a bottom-up style in which a program is written as a series of layers, each one acting as a sort of programming language for the one above. X Windows and TEX are examples of programs written in this style.

The theme of this book is twofold: that Lisp is a natural language for programs written in the bottom-up style, and that the bottom-up style is a natural way to write Lisp programs. *On Lisp* will thus be of interest to two classes of readers. For people interested in writing extensible programs, this book will show what you can do if you have the right language. For Lisp programmers, this book offers a practical explanation of how to use Lisp to its best advantage.

The title is intended to stress the importance of bottom-up programming in Lisp. Instead of just writing your program in Lisp, you can write your own language *on Lisp,* and write your program in that.

It is possible to write programs bottom-up in any language, but Lisp is the most natural vehicle for this style of programming. In Lisp, bottom-up design is not a special technique reserved for unusually large or difficult programs. Any substantial program will be written partly in this style. Lisp was meant from the start to be an extensible language. The language itself is mostly a collection of Lisp functions, no different from the ones you define yourself. What's more, Lisp functions can be expressed as lists, which are Lisp data structures. This means you can write Lisp functions which generate Lisp code.

A good Lisp programmer must know how to take advantage of this possibility. The usual way to do so is by defining a kind of operator called a *macro*. Mastering macros is one of the most important steps in moving from writing correct Lisp programs to writing beautiful ones. Introductory Lisp books have room for no more than a quick overview of macros: an explanation of what macros are, together with a few examples which hint at the strange and wonderful things you can do with them. Those strange and wonderful things will receive special attention here. One of the aims of this book is to collect in one place all that people have till now had to learn from experience about macros.

Understandably, introductory Lisp books do not emphasize the differences between Lisp and other languages. They have to get their message across to students who have, for the most part, been schooled to think of programs in Pascal terms. It would only confuse matters to explain that, while `defun` *looks* like a procedure definition, it is actually a program-writing program that generates code which builds a functional object and indexes it under the symbol given as the first argument.

One of the purposes of this book is to explain what makes Lisp different from other languages. When I began, I knew that, all other things being equal, I would much rather write programs in Lisp than in C or Pascal or Fortran. I knew also that this was not merely a question of taste. But I realized that if I was actually going to claim that Lisp was in some ways a better language, I had better be prepared to explain why.

When someone asked Louis Armstrong what jazz was, he replied "If you have to ask what jazz is, you'll never know." But he did answer the question in a way: he *showed* people what jazz was. That's one way to explain the power of Lisp—to demonstrate techniques that would be difficult or impossible in other languages. Most books on programming—even books on Lisp programming—deal with the kinds of programs you could write in any language. *On Lisp* deals mostly with the kinds of programs you could only write in Lisp. Extensibility, bottom-up programming, interactive development, source code transformation, embedded languages—this is where Lisp shows to advantage.

In principle, of course, any Turing-equivalent programming language can do the same things as any other. But that kind of power is not what programming languages are about. In principle, anything you can do with a programming language you can do with a Turing machine; in practice, programming a Turing machine is not worth the trouble.

So when I say that this book is about how to do things that are impossible in other languages, I don't mean "impossible" in the mathematical sense, but in the sense that matters for programming languages. That is, if you had to write some of the programs in this book in C, you might as well do it by writing a Lisp compiler in C first. Embedding Prolog in C, for example—can you imagine the amount of work that would take? Chapter 24 shows how to do it in 180 lines of Lisp.

I hoped to do more than simply demonstrate the power of Lisp, though. I also wanted to explain *why* Lisp is different. This turns out to be a subtle question—too subtle to be answered with phrases like "symbolic computation." What I have learned so far, I have tried to explain as clearly as I can.

Plan of the Book

Since functions are the foundation of Lisp programs, the book begins with several chapters on functions. Chapter 2 explains what Lisp functions are and the possibilities they offer. Chapter 3 then discusses the advantages of functional programming, the dominant style in Lisp programs. Chapter 4 shows how to use functions to extend Lisp. Then Chapter 5 suggests the new kinds of abstractions we can define with functions that return other functions. Finally, Chapter 6 shows how to use functions in place of traditional data structures.

The remainder of the book deals more with macros than functions. Macros receive more attention partly because there is more to say about them, and partly because they have not till now been adequately described in print. Chapters 7–10 form a complete tutorial on macro technique. By the end of it you will know most of what an experienced Lisp programmer knows about macros: how they work; how to define, test, and debug them; when to use macros and when not; the major types of macros; how to write programs which generate macro expansions; how macro style differs from Lisp style in general; and how to detect and cure each of the unique problems that afflict macros.

Following this tutorial, Chapters 11–18 show some of the powerful abstractions you can build with macros. Chapter 11 shows how to write the classic macros—those which create context, or implement loops or conditionals. Chapter 12 explains the role of macros in operations on generalized variables. Chapter 13 shows how macros can make programs run faster by shifting computation to compile-time. Chapter 14 introduces anaphoric macros, which allow you to use pronouns in your programs. Chapter 15 shows how macros provide a more convenient interface to the function-builders defined in Chapter 5. Chapter 16 shows how to use macro-defining macros to make Lisp write your programs for you. Chapter 17 discusses read-macros, and Chapter 18, macros for destructuring.

With Chapter 19 begins the fourth part of the book, devoted to embedded languages. Chapter 19 introduces the subject by showing the same program, a program to answer queries on a database, implemented first by an interpreter and then as a true embedded language. Chapter 20 shows how to introduce into Common Lisp programs the notion of a continuation, an object representing the remainder of a computation. Continuations are a very powerful tool, and can be used to implement both multiple processes and nondeterministic choice. Embedding these control structures in Lisp is discussed in Chapters 21 and 22, respectively. Nondeterminism, which allows you to write programs as if they had foresight, sounds like an abstraction of unusual power. Chapters 23 and 24 present two embedded languages which show that nondeterminism lives up to its promise: a complete ATN parser and an embedded Prolog which combined total ○ about 200 lines of code.

The fact that these programs are short means nothing in itself. If you resorted to writing incomprehensible code, there's no telling what you could do in 200 lines. The point is, these programs are not short because they depend on programming tricks, but because they're written using Lisp the way it's meant to be used. The point of Chapters 23 and 24 is not how to implement ATNs in one page of code or Prolog in two, but to show that these programs, when given their most natural Lisp implementation, simply are that short. The embedded languages in the latter chapters provide a proof by example of the twin points with which I began: that Lisp is a natural language for bottom-up design, and that bottom-up design is a natural way to use Lisp.

The book concludes with a discussion of object-oriented programming, and particularly CLOS, the Common Lisp Object System. By saving this topic till last, we see more clearly the way in which object-oriented programming is an extension of ideas already present in Lisp. It is one of the many abstractions that can be built *on Lisp*.

A chapter's worth of notes begins on page 387. The notes contain references, additional or alternative code, or descriptions of aspects of Lisp not directly related to the point at hand. Notes are indicated by a small circle in the outside margin, like this. There is also an Appendix (page 381) on packages. ○

Just as a tour of New York could be a tour of most of the world's cultures, a study of Lisp as the programmable programming language draws in most of Lisp technique. Most of the techniques described here are generally known in the Lisp community, but many have not till now been written down anywhere. And some issues, such as the proper role of macros or the nature of variable capture, are only vaguely understood even by many experienced Lisp programmers.

Examples

Lisp is a family of languages. Since Common Lisp promises to remain a widely used dialect, most of the examples in this book are in Common Lisp. The language was originally defined in 1984 by the publication of Guy Steele's *Common Lisp: the Language* (CLTL1). This definition was superseded in 1990 by the publication of the second edition (CLTL2), which will in turn yield place to the forthcoming ANSI standard. ○

This book contains hundreds of examples, ranging from single expressions to a working Prolog implementation. The code in this book has, wherever possible, been written to work in any version of Common Lisp. Those few examples which need features not found in CLTL1 implementations are explicitly identified in the text. Later chapters contain some examples in Scheme. These too are clearly identified.

The code is available by anonymous FTP from `endor.harvard.edu`, where it's in the directory `pub/onlisp`. Questions and comments can be sent to `onlisp@das.harvard.edu`.

Acknowledgements

While writing this book I have been particularly thankful for the help of Robert Morris. I went to him constantly for advice and was always glad I did. Several of the examples in this book are derived from code he originally wrote, including the version of `for` on page 127, the version of `aand` on page 191, `match` on page 239, the breadth-first `true-choose` on page 304, and the Prolog interpreter in Section 24.2. In fact, the whole book reflects (sometimes, indeed, transcribes) conversations I've had with Robert during the past seven years. (Thanks, rtm!)

I would also like to give special thanks to David Moon, who read large parts of the manuscript with great care, and gave me very useful comments. Chapter 12 was completely rewritten at his suggestion, and the example of variable capture on page 119 is one that he provided.

I was fortunate to have David Touretzky and Skona Brittain as the technical reviewers for the book. Several sections were added or rewritten at their suggestion. The alternative true nondeterministic choice operator on page 397 is based on a suggestion by David Toureztky.

Several other people consented to read all or part of the manuscript, including Tom Cheatham, Richard Draves (who also rewrote `alambda` and `propmacro` back in 1985), John Foderaro, David Hendler, George Luger, Robert Muller, Mark Nitzberg, and Guy Steele.

I'm grateful to Professor Cheatham, and Harvard generally, for providing the facilities used to write this book. Thanks also to the staff at Aiken Lab, including Tony Hartman, Janusz Juda, Harry Bochner, and Joanne Klys.

The people at Prentice Hall did a great job. I feel fortunate to have worked with Alan Apt, a good editor and a good guy. Thanks also to Mona Pompili, Shirley Michaels, and Shirley McGuire for their organization and good humor.

The incomparable Gino Lee of the Bow and Arrow Press, Cambridge, did the cover. The tree on the cover alludes specifically to the point made on page 27.

This book was typeset using LaTeX, a language written by Leslie Lamport atop Donald Knuth's TeX, with additional macros by L. A. Carr, Van Jacobson, and Guy Steele. The diagrams were done with Idraw, by John Vlissides and Scott Stanton. The whole was previewed with Ghostview, by Tim Theisen, which is built on Ghostscript, by L. Peter Deutsch. Gary Bisbee of Chiron Inc. produced the camera-ready copy.

I owe thanks to many others, including Paul Becker, Phil Chapnick, Alice Hartley, Glenn Holloway, Meichun Hsu, Krzysztof Lenk, Arman Maghbouleh, Howard Mullings, Nancy Parmet, Robert Penny, Gary Sabot, Patrick Slaney, Steve Strassman, Dave Watkins, the Weickers, and Bill Woods.

Most of all, I'd like to thank my parents, for their example and encouragement; and Jackie, who taught me what I might have learned if I had listened to them.

I hope reading this book will be fun. Of all the languages I know, I like Lisp the best, simply because it's the most beautiful. This book is about Lisp at its lispiest. I had fun writing it, and I hope that comes through in the text.

Paul Graham

Contents

1

The Extensible Language

Not long ago, if you asked what Lisp was for, many people would have answered "for artificial intelligence." In fact, the association between Lisp and AI is just an accident of history. Lisp was invented by John McCarthy, who also invented the term "artificial intelligence." His students and colleagues wrote their programs in Lisp, and so it began to be spoken of as an AI language. This line was taken up and repeated so often during the brief AI boom in the 1980s that it became almost an institution.

Fortunately, word has begun to spread that AI is not what Lisp is all about. Recent advances in hardware and software have made Lisp commercially viable: it is now used in Gnu Emacs, the best Unix text-editor; Autocad, the industry standard desktop CAD program; and Interleaf, a leading high-end publishing program. The way Lisp is used in these programs has nothing whatever to do with AI.

If Lisp is not the language of AI, what is it? Instead of judging Lisp by the company it keeps, let's look at the language itself. What can you do in Lisp that you can't do in other languages? One of the most distinctive qualities of Lisp is the way it can be tailored to suit the program being written in it. Lisp itself is a Lisp program, and Lisp programs can be expressed as lists, which are Lisp data structures. Together, these two principles mean that any user can add operators to Lisp which are indistinguishable from the ones that come built-in.

1.1 Design by Evolution

Because Lisp gives you the freedom to define your own operators, you can mold it into just the language you need. If you're writing a text-editor, you can turn

Lisp into a language for writing text-editors. If you're writing a CAD program, you can turn Lisp into a language for writing CAD programs. And if you're not sure yet what kind of program you're writing, it's a safe bet to write it in Lisp. Whatever kind of program yours turns out to be, Lisp will, during the writing of it, have evolved into a language for writing *that* kind of program.

If you're not sure yet what kind of program you're writing? To some ears that sentence has an odd ring to it. It is in jarring contrast with a certain model of doing things wherein you (1) carefully plan what you're going to do, and then (2) do it. According to this model, if Lisp encourages you to start writing your program before you've decided how it should work, it merely encourages sloppy thinking.

Well, it just ain't so. The plan-and-implement method may have been a good way of building dams or launching invasions, but experience has not shown it to be as good a way of writing programs. Why? Perhaps it's because computers are so exacting. Perhaps there is more variation between programs than there is between dams or invasions. Or perhaps the old methods don't work because old concepts of redundancy have no analogue in software development: if a dam contains 30% too much concrete, that's a margin for error, but if a program does 30% too much work, that *is* an error.

It may be difficult to say why the old method fails, but that it does fail, anyone can see. When is software delivered on time? Experienced programmers know that no matter how carefully you plan a program, when you write it the plans will turn out to be imperfect in some way. Sometimes the plans will be hopelessly wrong. Yet few of the victims of the plan-and-implement method question its basic soundness. Instead they blame human failings: if only the plans had been made with more foresight, all this trouble could have been avoided. Since even the very best programmers run into problems when they turn to implementation, perhaps it's too much to hope that people will ever have *that* much foresight. Perhaps the plan-and-implement method could be replaced with another approach which better suits our limitations.

We can approach programming in a different way, if we have the right tools. Why do we plan before implementing? The big danger in plunging right into a project is the possibility that we will paint ourselves into a corner. If we had a more flexible language, could this worry be lessened? We do, and it is. The flexibility of Lisp has spawned a whole new style of programming. In Lisp, you can do much of your planning as you write the program.

Why wait for hindsight? As Montaigne found, nothing clarifies your ideas like trying to write them down. Once you're freed from the worry that you'll paint yourself into a corner, you can take full advantage of this possibility. The ability to plan programs as you write them has two momentous consequences: programs take less time to write, because when you plan and write at the same time, you have a real program to focus your attention; and they turn out *better*, because the final design is always a product of evolution. So long as you maintain a certain discipline while searching for your program's destiny—so long as you always rewrite mistaken parts as soon as it becomes clear that they're mistaken—the final product will be a program more elegant than if you had spent weeks planning it beforehand.

Lisp's versatility makes this kind of programming a practical alternative. Indeed, the greatest danger of Lisp is that it may spoil you. Once you've used Lisp for a while, you may become so sensitive to the fit between language and application that you won't be able to go back to another language without always feeling that it doesn't give you quite the flexibility you need.

1.2 Programming Bottom-Up

It's a long-standing principle of programming style that the functional elements of a program should not be too large. If some component of a program grows beyond the stage where it's readily comprehensible, it becomes a mass of complexity which conceals errors as easily as a big city conceals fugitives. Such software will be hard to read, hard to test, and hard to debug.

In accordance with this principle, a large program must be divided into pieces, and the larger the program, the more it must be divided. How do you divide a program? The traditional approach is called *top-down design:* you say "the purpose of the program is to do these seven things, so I divide it into seven major subroutines. The first subroutine has to do these four things, so it in turn will have four of its own subroutines," and so on. This process continues until the whole program has the right level of granularity—each part large enough to do something substantial, but small enough to be understood as a single unit.

Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called *bottom-up design*—changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

It's worth emphasizing that bottom-up design doesn't mean just writing the same program in a different order. When you work bottom-up, you usually end up with a different program. Instead of a single, monolithic program, you will get a larger language with more abstract operators, and a smaller program written in it. Instead of a lintel, you'll get an arch.

In typical code, once you abstract out the parts which are merely bookkeeping, what's left is much shorter; the higher you build up the language, the less distance you will have to travel from the top down to it. This brings several advantages:

1. By making the language do more of the work, bottom-up design yields programs which are smaller and more agile. A shorter program doesn't have to be divided into so many components, and fewer components means programs which are easier to read or modify. Fewer components also means fewer connections between components, and thus less chance for errors there. As industrial designers strive to reduce the number of moving parts in a machine, experienced Lisp programmers use bottom-up design to reduce the size and complexity of their programs.

2. Bottom-up design promotes code re-use. When you write two or more programs, many of the utilities you wrote for the first program will also be useful in the succeeding ones. Once you've acquired a large substrate of utilities, writing a new program can take only a fraction of the effort it would require if you had to start with raw Lisp.

3. Bottom-up design makes programs easier to read. An instance of this type of abstraction asks the reader to understand a general-purpose operator; an instance of functional abstraction asks the reader to understand a special-purpose subroutine.[1]

4. Because it causes you always to be on the lookout for patterns in your code, working bottom-up helps to clarify your ideas about the design of your program. If two distant components of a program are similar in form, you'll be led to notice the similarity and perhaps to redesign the program in a simpler way.

Bottom-up design is possible to a certain degree in languages other than Lisp. Whenever you see library functions, bottom-up design is happening. However, Lisp gives you much broader powers in this department, and augmenting the language plays a proportionately larger role in Lisp style—so much so that Lisp is not just a different language, but a whole different way of programming.

[1] "But no one can read the program without understanding all your new utilities." To see why such statements are usually mistaken, see Section 4.8.

It's true that this style of development is better suited to programs which can be written by small groups. However, at the same time, it extends the limits of what can be done by a small group. In *The Mythical Man-Month*, Frederick Brooks proposed that the productivity of a group of programmers does not grow linearly with its size. As the size of the group increases, the productivity of individual programmers goes down. The experience of Lisp programming suggests a more cheerful way to phrase this law: as the size of the group decreases, the productivity of individual programmers goes up. A small group wins, relatively speaking, simply because it's smaller. When a small group also takes advantage of the techniques that Lisp makes possible, it can win outright.

1.3 Extensible Software

The Lisp style of programming is one that has grown in importance as software has grown in complexity. Sophisticated users now demand so much from software that we can't possibly anticipate all their needs. They themselves can't anticipate all their needs. But if we can't give them software which does everything they want right out of the box, we can give them software which is extensible. We transform our software from a mere program into a programming language, and advanced users can build upon it the extra features that they need.

Bottom-up design leads naturally to extensible programs. The simplest bottom-up programs consist of two layers: language and program. Complex programs may be written as a series of layers, each one acting as a programming language for the one above. If this philosophy is carried all the way up to the topmost layer, that layer becomes a programming language for the user. Such a program, where extensibility permeates every level, is likely to make a much better programming language than a system which was written as a traditional black box, and then made extensible as an afterthought.

X Windows and TeX are early examples of programs based on this principle. In the 1980s better hardware made possible a new generation of programs which had Lisp as their extension language. The first was Gnu Emacs, the popular Unix text-editor. Later came Autocad, the first large-scale commercial product to provide Lisp as an extension language. In 1991 Interleaf released a new version of its software that not only had Lisp as an extension language, but was largely implemented in Lisp.

Lisp is an especially good language for writing extensible programs because it is itself an extensible program. If you write your Lisp programs so as to pass this extensibility on to the user, you effectively get an extension language for free. And the difference between extending a Lisp program in Lisp, and doing the same thing in a traditional language, is like the difference between meeting someone in person and conversing by letters. In a program which is made extensible simply by providing access to outside programs, the best we can hope for is two black boxes communicating with one another through some predefined channel. In Lisp, extensions can have direct access to the entire underlying program. This is not to say that you have to give users access to every part of your program—just that you now have a *choice* about whether to give them access or not.

When this degree of access is combined with an interactive environment, you have extensibility at its best. Any program that you might use as a foundation for extensions of your own is likely to be fairly big—too big, probably, for you to have a complete mental picture of it. What happens when you're unsure of something? If the original program is written in Lisp, you can probe it interactively: you can inspect its data structures; you can call its functions; you may even be able to look at the original source code. This kind of feedback allows you to program with a high degree of confidence—to write more ambitious extensions, and to write them faster. An interactive environment always makes programming easier, but it is nowhere more valuable than when one is writing extensions.

An extensible program is a double-edged sword, but recent experience has shown that users prefer a double-edged sword to a blunt one. Extensible programs seem to prevail, whatever their inherent dangers.

1.4 Extending Lisp

There are two ways to add new operators to Lisp: functions and macros. In Lisp, functions you define have the same status as the built-in ones. If you want a new variant of `mapcar`, you can define one yourself and use it just as you would use `mapcar`. For example, if you want a list of the values returned by some function when it is applied to all the integers from `1` to `10`, you could create a new list and pass it to `mapcar`:

```
(mapcar fn
        (do* ((x 1 (1+ x))
              (result (list x) (push x result)))
            ((= x 10) (nreverse result))))
```

but this approach is both ugly and inefficient.[2] Instead you could define a new mapping function `map1-n` (see page 54), and then call it as follows:

```
(map1-n fn 10)
```

[2] You could write this more elegantly with the new Common Lisp series macros, but that only proves the same point, because these macros are an extension to Lisp themselves.

Defining functions is comparatively straightforward. Macros provide a more general, but less well-understood, means of defining new operators. Macros are programs that write programs. This statement has far-reaching implications, and exploring them is one of the main purposes of this book.

The thoughtful use of macros leads to programs which are marvels of clarity and elegance. These gems are not to be had for nothing. Eventually macros will seem the most natural thing in the world, but they can be hard to understand at first. Partly this is because they are more general than functions, so there is more to keep in mind when writing them. But the main reason macros are hard to understand is that they're *foreign.* No other language has anything like Lisp macros. Thus learning about macros may entail unlearning preconceptions inadvertently picked up from other languages. Foremost among these is the notion of a program as something afflicted by rigor mortis. Why should data structures be fluid and changeable, but programs not? In Lisp, programs *are* data, but the implications of this fact take a while to sink in.

If it takes some time to get used to macros, it is well worth the effort. Even in such mundane uses as iteration, macros can make programs significantly smaller and cleaner. Suppose a program must iterate over some body of code for `x` from `a` to `b`. The built-in Lisp `do` is meant for more general cases. For simple iteration it does not yield the most readable code:

```
(do ((x a (+ 1 x)))
    ((> x b))
  (print x))
```

Instead, suppose we could just say:

```
(for (x a b)
  (print x))
```

Macros make this possible. With six lines of code (see page 154) we can add `for` to the language, just as if it had been there from the start. And as later chapters will show, writing `for` is only the beginning of what you can do with macros.

You're not limited to extending Lisp one function or macro at a time. If you need to, you can build a whole language on top of Lisp, and write your programs in that. Lisp is an excellent language for writing compilers and interpreters, but it offers another way of defining a new language which is often more elegant and certainly much less work: to define the new language as a modification of Lisp. Then the parts of Lisp which can appear unchanged in the new language (e.g. arithmetic or I/O) can be used as is, and you only have to implement the parts which are different (e.g. control structure). A language implemented in this way is called an *embedded language.*

Embedded languages are a natural outgrowth of bottom-up programming. Common Lisp includes several already. The most famous of them, CLOS, is discussed in the last chapter. But you can define embedded languages of your own, too. You can have the language which suits your program, even if it ends up looking quite different from Lisp.

1.5 Why Lisp (or When)

These new possibilities do not stem from a single magic ingredient. In this respect, Lisp is like an arch. Which of the wedge-shaped stones (voussoirs) is the one that holds up the arch? The question itself is mistaken; they all do. Like an arch, Lisp is a collection of interlocking features. We can list some of these features—dynamic storage allocation and garbage collection, runtime typing, functions as objects, a built-in parser which generates lists, a compiler which accepts programs expressed as lists, an interactive environment, and so on—but the power of Lisp cannot be traced to any single one of them. It is the combination which makes Lisp programming what it is.

Over the past twenty years, the way people program has changed. Many of these changes—interactive environments, dynamic linking, even object-oriented programming—have been piecemeal attempts to give other languages some of the flexibility of Lisp. The metaphor of the arch suggests how well they have succeeded.

It is widely known that Lisp and Fortran are the two oldest languages still in use. What is perhaps more significant is that they represent opposite poles in the philosophy of language design. Fortran was invented as a step up from assembly language. Lisp was invented as a language for expressing algorithms. Such different intentions yielded vastly different languages. Fortran makes life easy for the compiler writer; Lisp makes life easy for the programmer. Most programming languages since have fallen somewhere between the two poles. Fortran and Lisp have themselves moved closer to the center. Fortran now looks more like Algol, and Lisp has given up some of the wasteful habits of its youth.

The original Fortran and Lisp defined a sort of battlefield. On one side the battle cry is "Efficiency! (And besides, it would be too hard to implement.)" On the other side, the battle cry is "Abstraction! (And anyway, this isn't production software.)" As the gods determined from afar the outcomes of battles among the ancient Greeks, the outcome of this battle is being determined by hardware. Every year, things look better for Lisp. The arguments against Lisp are now starting to sound very much like the arguments that assembly language programmers gave against high-level languages in the early 1970s. The question is now becoming not *Why Lisp?*, but *When?*

2

Functions

Functions are the building-blocks of Lisp programs. They are also the building-blocks of Lisp. In most languages the + operator is something quite different from user-defined functions. But Lisp has a single model, function application, to describe all the computation done by a program. The Lisp + operator is a function, just like the ones you can define yourself.

In fact, except for a small number of operators called *special forms*, the core of Lisp is a collection of Lisp functions. What's to stop you from adding to this collection? Nothing at all: if you think of something you wish Lisp could do, you can write it yourself, and your new function will be treated just like the built-in ones.

This fact has important consequences for the programmer. It means that any new function could be considered either as an addition to Lisp, or as part of a specific application. Typically, an experienced Lisp programmer will write some of each, adjusting the boundary between language and application until the two fit one another perfectly. This book is about how to achieve a good fit between language and application. Since everything we do toward this end ultimately depends on functions, functions are the natural place to begin.

2.1 Functions as Data

Two things make Lisp functions different. One, mentioned above, is that Lisp itself is a collection of functions. This means that we can add to Lisp new operators of our own. Another important thing to know about functions is that they are Lisp objects.

Lisp offers most of the data types one finds in other languages. We get integers and floating-point numbers, strings, arrays, structures, and so on. But Lisp supports one data type which may at first seem surprising: the function. Nearly all programming languages provide some form of function or procedure. What does it mean to say that Lisp provides them as a data type? It means that in Lisp we can do with functions all the things we expect to do with more familiar data types, like integers: create new ones at runtime, store them in variables and in structures, pass them as arguments to other functions, and return them as results.

The ability to create and return functions at runtime is particularly useful. This might sound at first like a dubious sort of advantage, like the self-modifying machine language programs one can run on some computers. But creating new functions at runtime turns out to be a routinely used Lisp programming technique.

2.2 Defining Functions

Most people first learn how to make functions with `defun`. The following expression defines a function called `double` which returns twice its argument.

```
> (defun double (x) (* x 2))
DOUBLE
```

Having fed this to Lisp, we can call `double` in other functions, or from the toplevel:

```
> (double 1)
2
```

A file of Lisp code usually consists mainly of such `defun`s, and so resembles a file of procedure definitions in a language like C or Pascal. But something quite different is going on. Those `defun`s are not just procedure definitions, they're Lisp calls. This distinction will become clearer when we see what's going on underneath `defun`.

Functions are objects in their own right. What `defun` really does is build one, and store it under the name given as the first argument. So as well as calling `double`, we can get hold of the function which implements it. The usual way to do so is by using the `#'` (sharp-quote) operator. This operator can be understood as mapping names to actual function objects. By affixing it to the name of `double`

```
> #'double
#<Interpreted-Function C66ACE>
```

we get the actual object created by the definition above. Though its printed representation will vary from implementation to implementation, a Common Lisp function is a first-class object, with all the same rights as more familiar objects like numbers and strings. So we can pass this function as an argument, return it, store it in a data structure, and so on:

```
> (eq #'double (car (list #'double)))
T
```

We don't even need `defun` to make functions. Like most Lisp objects, we can refer to them literally. When we want to refer to an integer, we just use the integer itself. To represent a string, we use a series of characters surrounded by double-quotes. To represent a function, we use what's called a *lambda-expression.* A lambda-expression is a list with three parts: the symbol `lambda`, a parameter list, and a body of zero or more expressions. This lambda-expression refers to a function equivalent to `double`:

```
(lambda (x) (* x 2))
```

It describes a function which takes one argument *x*, and returns *2x*.

A lambda-expression can also be considered as the name of a function. If `double` is a proper name, like "Michelangelo," then `(lambda (x) (* x 2))` is a definite description, like "the man who painted the ceiling of the Sistine Chapel." By putting a sharp-quote before a lambda-expression, we get the corresponding function:

```
> #'(lambda (x) (* x 2))
#<Interpreted-Function C674CE>
```

This function behaves exactly like `double`, but the two are distinct objects.

In a function call, the name of the function appears first, followed by the arguments:

```
> (double 3)
6
```

Since lambda-expressions are also names of functions, they can also appear first in function calls:

```
> ((lambda (x) (* x 2)) 3)
6
```

In Common Lisp, we can have a function named `double` and a variable named `double` at the same time.

```
> (setq double 2)
2
> (double double)
4
```

When a name occurs first in a function call, or is preceded by a sharp-quote, it is taken to refer to a function. Otherwise it is treated as a variable name.

It is therefore said that Common Lisp has distinct *name-spaces* for variables and functions. We can have a variable called `foo` and a function called `foo`, and they need not be identical. This situation can be confusing, and leads to a certain amount of ugliness in code, but it is something that Common Lisp programmers have to live with.

If necessary, Common Lisp provides two functions which map symbols to the values, or functions, that they represent. The function `symbol-value` takes a symbol and returns the value of the corresponding special variable:

```
> (symbol-value 'double)
2
```

while `symbol-function` does the same for a globally defined function:

```
> (symbol-function 'double)
#<Interpreted-Function C66ACE>
```

Note that, since functions are ordinary data objects, a variable could have a function as its value:

```
> (setq x #'append)
#<Compiled-Function 46B4BE>
> (eq (symbol-value 'x) (symbol-function 'append))
T
```

Beneath the surface, `defun` is setting the `symbol-function` of its first argument to a function constructed from the remaining arguments. The following two expressions do approximately the same thing:

```
(defun double (x) (* x 2))

(setf (symbol-function 'double)
      #'(lambda (x) (* x 2)))
```

So `defun` has the same effect as procedure definition in other languages—to associate a name with a piece of code. But the underlying mechanism is not the same. We don't need `defun` to make functions, and functions don't have to be stored away as the value of some symbol. Underlying `defun`, which resembles procedure definition in any other language, is a more general mechanism: building a function and associating it with a certain name are two separate operations. When we don't need the full generality of Lisp's notion of a function, `defun` makes function definition as simple as in more restrictive languages.

2.3 Functional Arguments

Having functions as data objects means, among other things, that we can pass them as arguments to other functions. This possibility is partly responsible for the importance of bottom-up programming in Lisp.

A language which allows functions as data objects must also provide some way of calling them. In Lisp, this function is `apply`. Generally, we call `apply` with two arguments: a function, and a list of arguments for it. The following four expressions all have the same effect:

```
(+ 1 2)

(apply #'+ '(1 2))

(apply (symbol-function '+) '(1 2))

(apply #'(lambda (x y) (+ x y)) '(1 2))
```

In Common Lisp, `apply` can take any number of arguments, and the function given first will be applied to the list made by consing the rest of the arguments onto the list given last. So the expression

```
(apply #'+ 1 '(2))
```

is equivalent to the preceding four. If it is inconvenient to give the arguments as a list, we can use `funcall`, which differs from `apply` only in this respect. This expression

```
(funcall #'+ 1 2)
```

has the same effect as those above.

Many built-in Common Lisp functions take functional arguments. Among the most frequently used are the mapping functions. For example, `mapcar` takes two or more arguments, a function and one or more lists (one for each parameter of the function), and applies the function successively to elements of each list:

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)
> (mapcar #'+
          '(1 2 3)
          '(10 100 1000))
(11 102 1003)
```

Lisp programs frequently want to do something to each element of a list and get back a list of results. The first example above illustrates the conventional way to do this: make a function which does what you want done, and mapcar it over the list.

Already we see how convenient it is to be able to treat functions as data. In many languages, even if we could pass a function as an argument to something like `mapcar`, it would still have to be a function defined in some source file beforehand. If just one piece of code wanted to add 10 to each element of a list, we would have to define a function, called `plus_ten` or some such, just for this one use. With lambda-expressions, we can refer to functions directly.

One of the big differences between Common Lisp and the dialects which preceded it are the large number of built-in functions that take functional arguments. Two of the most commonly used, after the ubiquitous `mapcar`, are `sort` and `remove-if`. The former is a general-purpose sorting function. It takes a list and a predicate, and returns a list sorted by passing each pair of elements to the predicate.

```
> (sort '(1 4 2 5 6 7 3) #'<)
(1 2 3 4 5 6 7)
```

To remember how `sort` works, it helps to remember that if you sort a list with no duplicates by <, and then apply < to the resulting list, it will return true.

If `remove-if` weren't included in Common Lisp, it might be the first utility you would write. It takes a function and a list, and returns all the elements of the list for which the function returns false.

```
> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)
```

As an example of a function which takes functional arguments, here is a definition of a limited version of `remove-if`:

```
(defun our-remove-if (fn lst)
  (if (null lst)
      nil
      (if (funcall fn (car lst))
          (our-remove-if fn (cdr lst))
          (cons (car lst) (our-remove-if fn (cdr lst))))))
```

Note that within this definition `fn` is not sharp-quoted. Since functions are data objects, a variable can have a function as its regular value. That's what's happening here. Sharp-quote is only for referring to the function named by a symbol—usually one globally defined as such with `defun`.

As Chapter 4 will show, writing new utilities which take functional arguments is an important element of bottom-up programming. Common Lisp has so many utilities built-in that the one you need may exist already. But whether you use built-ins like `sort`, or write your own utilities, the principle is the same. Instead of wiring in functionality, pass a functional argument.

2.4 Functions as Properties

The fact that functions are Lisp objects also allows us to write programs which can be extended to deal with new cases on the fly. Suppose we want to write a function which takes a type of animal and behaves appropriately. In most languages, the way to do this would be with a `case` statement, and we can do it this way in Lisp as well:

```
(defun behave (animal)
  (case animal
    (dog (wag-tail)
         (bark))
    (rat (scurry)
         (squeak))
    (cat (rub-legs)
         (scratch-carpet))))
```

What if we want to add a new type of animal? If we were planning to add new animals, it would have been better to define `behave` as follows:

```
(defun behave (animal)
  (funcall (get animal 'behavior)))
```

and to define the behavior of an individual animal as a function stored, for example, on the property list of its name:

```
(setf (get 'dog 'behavior)
      #'(lambda ()
          (wag-tail)
          (bark)))
```

This way, all we need do in order to add a new animal is define a new property. No functions have to be rewritten.

The second approach, though more flexible, looks slower. It is. If speed were critical, we would use structures instead of property lists and, especially, compiled instead of interpreted functions. (Section 2.9 explains how to make these.) With structures and compiled functions, the more flexible type of code can approach or exceed the speed of versions using `case` statements.

This use of functions corresponds to the concept of a *method* in object-oriented programming. Generally speaking, a method is a function which is a property of an object, and that's just what we have. If we add inheritance to this model, we'll have all the elements of object-oriented programming. Chapter 25 will show that this can be done with surprisingly little code.

One of the big selling points of object-oriented programming is that it makes programs extensible. This prospect excites less wonder in the Lisp world, where extensibility has always been taken for granted. If the kind of extensibility we need does not depend too much on inheritance, then plain Lisp may already be sufficient.

2.5 Scope

Common Lisp is a lexically scoped Lisp. Scheme is the oldest dialect with lexical scope; before Scheme, dynamic scope was considered one of the defining features of Lisp.

The difference between lexical and dynamic scope comes down to how an implementation deals with free variables. A symbol is *bound* in an expression if it has been established as a variable, either by appearing as a parameter, or by variable-binding operators like `let` and `do`. Symbols which are not bound are said to be *free*. In this example, scope comes into play:

```
(let ((y 7))
  (defun scope-test (x)
    (list x y)))
```

Within the `defun` expression, `x` is bound and `y` is free. Free variables are interesting because it's not obvious what their values should be. There's no uncertainty about the value of a bound variable—when `scope-test` is called, the value of `x` should be whatever is passed as the argument. But what should be the value of y? This is the question answered by the dialect's scope rules.

In a dynamically scoped Lisp, to find the value of a free variable when executing `scope-test`, we look back through the chain of functions that called it. When we find an environment where `y` was bound, that binding of `y` will be the one used in `scope-test`. If we find none, we take the global value of `y`. Thus, in a dynamically scoped Lisp, `y` would have the value it had in the calling expression:

```
> (let ((y 5))
    (scope-test 3))
(3 5)
```

With dynamic scope, it means nothing that `y` was bound to `7` when `scope-test` was defined. All that matters is that `y` had a value of `5` when `scope-test` was called.

In a lexically scoped Lisp, instead of looking back through the chain of calling functions, we look back through the containing environments at the time the function was *defined*. In a lexically scoped Lisp, our example would catch the binding of `y` where `scope-test` was defined. So this is what would happen in Common Lisp:

```
> (let ((y 5))
    (scope-test 3))
(3 7)
```

Here the binding of `y` to `5` at the time of the call has no effect on the returned value.

Though you can still get dynamic scope by declaring a variable to be `special`, lexical scope is the default in Common Lisp. On the whole, the Lisp community seems to view the passing of dynamic scope with little regret. For one thing, it used to lead to horribly elusive bugs. But lexical scope is more than a way of avoiding bugs. As the next section will show, it also makes possible some new programming techniques.

2.6 Closures

Because Common Lisp is lexically scoped, when we define a function containing free variables, the system must save copies of the bindings of those variables at the time the function was defined. Such a combination of a function and a set of variable bindings is called a *closure*. Closures turn out to be useful in a wide variety of applications.

Closures are so pervasive in Common Lisp programs that it's possible to use them without even knowing it. Every time you give `mapcar` a sharp-quoted lambda-expression containing free variables, you're using closures. For example, suppose we want to write a function which takes a list of numbers and adds a certain amount to each one. The function `list+`

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

will do what we want:

```
> (list+ '(1 2 3) 10)
(11 12 13)
```

If we look closely at the function which is passed to `mapcar` within `list+`, it's actually a closure. The instance of `n` is free, and its binding comes from the surrounding environment. Under lexical scope, every such use of a mapping function causes the creation of a closure.[1]

Closures play a more conspicuous role in a style of programming promoted ○ by Abelson and Sussman's classic *Structure and Interpretation of Computer Programs.* Closures are functions with local state. The simplest way to use this state is in a situation like the following:

```
(let ((counter 0))
  (defun new-id ()   (incf counter))
  (defun reset-id () (setq counter 0)))
```

These two functions share a variable which serves as a counter. The first one returns successive values of the counter, and the second resets the counter to `0`. The same thing could be done by making the counter a global variable, but this way it is protected from unintended references.

It's also useful to be able to return functions with local state. For example, the function `make-adder`

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

takes a number, and returns a closure which, when called, adds that number to its argument. We can make as many instances of adders as we want:

[1]Under dynamic scope the same idiom will work for a different reason—so long as neither of `mapcar`'s parameter is called `x`.

```
> (setq add2  (make-adder 2)
        add10 (make-adder 10))
#<Interpreted-Function BF162E>
> (funcall add2 5)
7
> (funcall add10 3)
13
```

In the closures returned by `make-adder`, the internal state is fixed, but it's also possible to make closures which can be asked to change their state.

```
(defun make-adderb (n)
  #'(lambda (x &optional change)
      (if change
          (setq n x)
          (+ x n))))
```

This new version of `make-adder` returns closures which, when called with one argument, behave just like the old ones.

```
> (setq addx (make-adderb 1))
#<Interpreted-Function BF1C66>
> (funcall addx 3)
4
```

However, when the new type of adder is called with a non-nil second argument, its internal copy of `n` will be reset to the value passed as the first argument:

```
> (funcall addx 100 t)
100
> (funcall addx 3)
103
```

It's even possible to return a group of closures which share the same data objects. Figure 2.1 contains a function which creates primitive databases. It takes an assoc-list (`db`), and returns a list of three closures which query, add, and delete entries, respectively.

Each call to `make-dbms` makes a new database—a new set of functions closed over their own shared copy of an assoc-list.

```
> (setq cities (make-dbms '((boston . us) (paris . france))))
(#<Interpreted-Function 8022E7>
 #<Interpreted-Function 802317>
 #<Interpreted-Function 802347>)
```

```
(defun make-dbms (db)
  (list
    #'(lambda (key)
        (cdr (assoc key db)))
    #'(lambda (key val)
        (push (cons key val) db)
        key)
    #'(lambda (key)
        (setf db (delete key db :key #'car))
        key)))
```

Figure 2.1: Three closures share a list.

The actual assoc-list within the database is invisible from the outside world—we can't even tell that it's an assoc-list—but it can be reached through the functions which are components of `cities`:

```
> (funcall (car cities) 'boston)
US
> (funcall (second cities) 'london 'england)
LONDON
> (funcall (car cities) 'london)
ENGLAND
```

Calling the `car` of a list is a bit ugly. In real programs, the access functions might instead be entries in a structure. Using them could also be cleaner—databases could be reached indirectly via functions like:

```
(defun lookup (key db)
  (funcall (car db) key))
```

However, the basic behavior of closures is independent of such refinements.

In real programs, the closures and data structures would also be more elaborate than those we see in `make-adder` or `make-dbms`. The single shared variable could be any number of variables, each bound to any sort of data structure.

Closures are one of the distinct, tangible benefits of Lisp. Some Lisp programs could, with effort, be translated into less powerful languages. But just try to translate a program which uses closures as above, and it will become evident how much work this abstraction is saving us. Later chapters will deal with closures in more detail. Chapter 5 shows how to use them to build compound functions, and Chapter 6 looks at their use as a substitute for traditional data structures.

2.7 Local Functions

When we define functions with lambda-expressions, we face a restriction which doesn't arise with `defun`: a function defined in a lambda-expression doesn't have a name and therefore has no way of referring to itself. This means that in Common Lisp we can't use `lambda` to define a recursive function. ○

If we want to apply some function to all the elements of a list, we use the most familiar of Lisp idioms:

```
> (mapcar #'(lambda (x) (+ 2 x))
          '(2 5 7 3))
(4 7 9 5)
```

What about cases where we want to give a recursive function as the first argument to `mapcar`? If the function has been defined with `defun`, we can simply refer to it by name:

```
> (mapcar #'copy-tree '((a b) (c d e)))
((A B) (C D E))
```

But now suppose that the function has to be a closure, taking some bindings from the environment in which the `mapcar` occurs. In our example `list+`,

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

the first argument to `mapcar`, `#'(lambda (x) (+ x n))`, must be defined within `list+` because it needs to catch the binding of `n`. So far so good, but what if we want to give `mapcar` a function which both needs local bindings *and* is recursive? We can't use a function defined elsewhere with `defun`, because we need bindings from the local environment. And we can't use `lambda` to define a recursive function, because the function will have no way of referring to itself.

Common Lisp gives us `labels` as a way out of this dilemma. With one important reservation, `labels` could be described as a sort of `let` for functions. Each of the binding specifications in a `labels` expression should have the form

`(`⟨name⟩ ⟨parameters⟩ `.` ⟨body⟩`)`

Within the `labels` expression, ⟨name⟩ will refer to a function equivalent to:

`#'(lambda` ⟨parameters⟩ `.` ⟨body⟩`)`

So for example:

```
> (labels ((inc (x) (1+ x)))
    (inc 3))
4
```

However, there is an important difference between `let` and `labels`. In a `let` expression, the value of one variable can't depend on another variable made by the same `let`—that is, you can't say

```
(let ((x 10) (y x))
  y)
```

and expect the value of the new `y` to reflect that of the new `x`. In contrast, the body of a function *f* defined in a `labels` expression may refer to any other function defined there, including *f* itself, which makes recursive function definitions possible.

Using `labels` we can write a function analogous to `list+`, but in which the first argument to `mapcar` is a recursive function:

```
(defun count-instances (obj lsts)
  (labels ((instances-in (lst)
             (if (consp lst)
                 (+ (if (eq (car lst) obj) 1 0)
                    (instances-in (cdr lst)))
                 0)))
    (mapcar #'instances-in lsts)))
```

This function takes an object and a list, and returns a list of the number of occurrences of the object in each element:

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

2.8 Tail-Recursion

A recursive function is one that calls itself. Such a call is *tail-recursive* if no work remains to be done in the calling function afterwards. This function is not tail-recursive

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

because on returning from the recursive call we have to pass the result to `1+`. The following function is tail-recursive, though

```
(defun our-find-if (fn lst)
  (if (funcall fn (car lst))
      (car lst)
      (our-find-if fn (cdr lst))))
```

because the value of the recursive call is immediately returned.

Tail-recursion is desirable because many Common Lisp compilers can transform tail-recursive functions into loops. With such a compiler, you can have the elegance of recursion in your source code without the overhead of function calls at runtime. The gain in speed is usually great enough that programmers go out of their way to make functions tail-recursive.

A function which isn't tail-recursive can often be transformed into one that is by embedding in it a local function which uses an *accumulator*. In this context, an accumulator is a parameter representing the value computed so far. For example, `our-length` could be transformed into

```
(defun our-length (lst)
  (labels ((rec (lst acc)
             (if (null lst)
                 acc
                 (rec (cdr lst) (1+ acc)))))
    (rec lst 0)))
```

where the number of list elements seen so far is contained in a second parameter, `acc`. When the recursion reaches the end of the list, the value of `acc` will be the total length, which can just be returned. By accumulating the value as we go down the calling tree instead of constructing it on the way back up, we can make `rec` tail-recursive.

Many Common Lisp compilers can do tail-recursion optimization, but not all of them do it by default. So after writing your functions to be tail-recursive, you may also want to put

```
(proclaim '(optimize speed))
```

at the top of the file, to ensure that the compiler can take advantage of your efforts.[2]

Given tail-recursion and type declarations, existing Common Lisp compilers can generate code that runs as fast as, or faster than, C. Richard Gabriel gives as ◦
an example the following function, which returns the sum of the integers from `1` to `n`:

[2]The declaration `(optimize speed)` ought to be an abbreviation for `(optimize (speed 3))`. However, one Common Lisp implementation does tail-recursion optimization with the former, but not the latter.

```
(defun triangle (n)
  (labels ((tri (c n)
             (declare (type fixnum n c))
             (if (zerop n)
                 c
                 (tri (the fixnum (+ n c))
                      (the fixnum (- n 1))))))
    (tri 0 n)))
```

This is what fast Common Lisp code looks like. At first it may not seem natural to write functions this way. It's often a good idea to begin by writing a function in whatever way seems most natural, and then, if necessary, transforming it into a tail-recursive equivalent.

2.9 Compilation

Lisp functions can be compiled either individually or by the file. If you just type a `defun` expression into the toplevel,

```
> (defun foo (x) (1+ x))
FOO
```

many implementations will create an interpreted function. You can check whether a given function is compiled by feeding it to `compiled-function-p`:

```
> (compiled-function-p #'foo)
NIL
```

We can have `foo` compiled by giving its name to `compile`

```
> (compile 'foo)
FOO
```

which will compile the definition of `foo` and replace the interpreted version with ○ a compiled one.

```
> (compiled-function-p #'foo)
T
```

Compiled and interpreted functions are both Lisp objects, and behave the same, except with respect to `compiled-function-p`. Literal functions can also be compiled: `compile` expects its first argument to be a name, but if you give `nil` as the first argument, it will compile the lambda-expression given as the second argument.

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function BF55BE>
```

If you give both the name and function arguments, `compile` becomes a sort of compiling `defun`:

```
> (progn (compile 'bar '(lambda (x) (* x 3)))
         (compiled-function-p #'bar))
T
```

Having `compile` in the language means that a program could build and compile new functions on the fly. However, calling `compile` explicitly is a drastic measure, comparable to calling `eval`, and should be viewed with the same suspicion.[3] When Section 2.1 said that creating new functions at runtime was a routinely used programming technique, it referred to new closures like those made by `make-adder`, not functions made by calling `compile` on raw lists. Calling `compile` is not a routinely used programming technique—it's an extremely rare one. So beware of doing it unnecessarily. Unless you're implementing another language on top of Lisp (and much of the time, even then), what you need to do may be possible with macros.

There are two sorts of functions which you can't give as an argument to `compile`. According to CLTL2 (p. 677), you can't compile a function "defined interpretively in a non-null lexical environment." That is, if at the toplevel you define `foo` within a `let`

```
> (let ((y 2))
    (defun foo (x) (+ x y)))
```

then `(compile 'foo)` will not necessarily work.[4] You also can't call `compile` on a function which is already compiled. In this situation, CLTL2 hints darkly that "the consequences. . .are unspecified."

The usual way to compile Lisp code is not to compile functions individually with `compile`, but to compile whole files with `compile-file`. This function takes a filename and creates a compiled version of the source file—typically with the same base name but a different extension. When the compiled file is loaded, `compiled-function-p` should return true for all the functions defined in the file.

Later chapters will depend on another effect of compilation: when one function occurs within another function, and the containing function is compiled, the inner

---

[3] An explanation of why it is bad to call `eval` explicitly appears on page 278.

[4] It's ok to have this code in a file and then compile the file. The restriction is imposed on interpreted code for implementation reasons, not because there's anything wrong with defining functions in distinct lexical environments.

function will also get compiled. CLTL2 does not seem to say explicitly that this will happen, but in a decent implementation you can count on it.

The compiling of inner functions becomes evident in functions which return functions. When `make-adder` (page 18) is compiled, it will return compiled functions:

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-function-p (make-adder 2))
T
```

As later chapters will show, this fact is of great importance in the implementation of embedded languages. If a new language is implemented by transformation, and the transformation code is compiled, then it yields compiled output—and so becomes in effect a compiler for the new language. (A simple example is described on page 81.)

If we have a particularly small function, we may want to request that it be compiled inline. Otherwise, the machinery of calling it could entail more effort than the function itself. If we define a function:

```
(defun 50th (lst) (nth 49 lst))
```

and make the declaration:

```
(proclaim '(inline 50th))
```

then a reference to `50th` within a compiled function should no longer require a real function call. If we define and compile a function which calls `50th`,

```
(defun foo (lst)
  (+ (50th lst) 1))
```

then when `foo` is compiled, the code for `50th` should be compiled right into it, just as if we had written

```
(defun foo (lst)
  (+ (nth 49 lst) 1))
```

in the first place. The drawback is that if we redefine `50th`, we also have to recompile `foo`, or it will still reflect the old definition. The restrictions on inline ◦ functions are basically the same as those on macros (see Section 7.9).

2.10 Functions from Lists

In some earlier dialects of Lisp, functions were represented as lists. This gave Lisp programs the remarkable ability to write and execute their own Lisp programs. In Common Lisp, functions are no longer made of lists—good implementations compile them into native machine code. But you can still write programs that write programs, because lists are the input to the compiler.

It cannot be overemphasized how important it is that Lisp programs can write Lisp programs, especially since this fact is so often overlooked. Even experienced Lisp users rarely realize the advantages they derive from this feature of the language. *This* is why Lisp macros are so powerful, for example. Most of the techniques described in this book depend on the ability to write programs which manipulate Lisp expressions.

3

Functional Programming

The previous chapter explained how Lisp and Lisp programs are both built out of a single raw material: the function. Like any building material, its qualities influence both the kinds of things we build, and the way we build them.

This chapter describes the kind of construction methods which prevail in the Lisp world. The sophistication of these methods allows us to attempt more ambitious kinds of programs. The next chapter will describe one particularly important class of programs which become possible in Lisp: programs which evolve instead of being developed by the old plan-and-implement method.

3.1 Functional Design

The character of an object is influenced by the elements from which it is made. A wooden building looks different from a stone one, for example. Even when you are too far away to see wood or stone, you can tell from the overall shape of the building what it's made of. The character of Lisp functions has a similar influence on the structure of Lisp programs.

Functional programming means writing programs which work by returning values instead of by performing side-effects. Side-effects include destructive changes to objects (e.g. by `rplaca`) and assignments to variables (e.g. by `setq`). If side-effects are few and localized, programs become easier to read, test, and debug. Lisp programs have not always been written in this style, but over time Lisp and functional programming have gradually become inseparable.

An example will show how functional programming differs from what you might do in another language. Suppose for some reason we want the elements of

```
(defun bad-reverse (lst)
  (let* ((len (length lst))
         (ilimit (truncate (/ len 2))))
    (do ((i 0 (1+ i))
         (j (1- len) (1- j)))
        ((>= i ilimit))
      (rotatef (nth i lst) (nth j lst)))))
```

Figure 3.1: A function to reverse lists.

a list in the reverse order. Instead of writing a function to reverse lists, we write a function which takes a list, and returns a list with the same elements in the reverse order.

Figure 3.1 contains a function to reverse lists. It treats the list as an array, reversing it in place; its return value is irrelevant:

```
> (setq lst '(a b c))
(A B C)
> (bad-reverse lst)
NIL
> lst
(C B A)
```

As its name suggests, `bad-reverse` is far from good Lisp style. Moreover, its ugliness is contagious: because it works by side-effects, it will also draw its callers away from the functional ideal.

Though cast in the role of the villain, `bad-reverse` does have one merit: it shows the Common Lisp idiom for swapping two values. The `rotatef` macro rotates the values of any number of generalized variables—that is, expressions you could give as the first argument to `setf`. When applied to just two arguments, the effect is to swap them.

In contrast, Figure 3.2 shows a function which returns reversed lists. With `good-reverse`, we get the reversed list as the return value; the original list is not touched.

```
> (setq lst '(a b c))
(A B C)
> (good-reverse lst)
(C B A)
> lst
(A B C)
```

```
(defun good-reverse (lst)
  (labels ((rev (lst acc)
             (if (null lst)
                 acc
                 (rev (cdr lst) (cons (car lst) acc)))))
    (rev lst nil)))
```

Figure 3.2: A function to return reversed lists.

It used to be thought that you could judge someone's character by looking at the shape of his head. Whether or not this is true of people, it is generally true of Lisp programs. Functional programs have a different shape from imperative ones. The structure in a functional program comes entirely from the composition of arguments within expressions, and since arguments are indented, functional code will show more variation in indentation. Functional code looks fluid[1] on the page; imperative code looks solid and blockish, like Basic.

Even from a distance, the shapes of `bad-` and `good-reverse` suggest which is the better program. And despite being shorter, `good-reverse` is also more efficient: $O(n)$ instead of $O(n^2)$.

We are spared the trouble of writing `reverse` because Common Lisp has it built-in. It is worth looking briefly at this function, because it is one that often brings to the surface misconceptions about functional programming. Like `good-reverse`, the built-in `reverse` works by returning a value—it doesn't touch its arguments. But people learning Lisp may assume that, like `bad-reverse`, it works by side-effects. If in some part of a program they want a list `lst` to be reversed, they may write

```
(reverse lst)
```

and wonder why the call seems to have no effect. In fact, if we want *effects* from such a function, we have to see to it ourselves in the calling code. That is, we need to write

```
(setq lst (reverse lst))
```

instead. Operators like `reverse` are intended to be called for return values, not side-effects. It is worth writing your own programs in this style too—not only for its inherent benefits, but because, if you don't, you will be working against the language.

[1]For a characteristic example, see page 242.

One of the points we ignored in the comparison of `bad-` and `good-reverse` is that `bad-reverse` doesn't cons. Instead of building new list structure, it operates on the original list. This can be dangerous—the list could be needed elsewhere in the program—but for efficiency it is sometimes necessary. For such cases, Common Lisp provides an *O(n)* destructive reversing function called `nreverse`. ○

A destructive function is one that can alter the arguments passed to it. However, even destructive functions usually work by returning values: you have to assume that `nreverse` will recycle lists you give to it as arguments, but you still can't assume that it will reverse them. As before, the reversed list has to be found in the return value. You still can't write

```
(nreverse lst)
```

in the middle of a function and assume that afterwards `lst` will be reversed. This is what happens in most implementations:

```
> (setq lst '(a b c))
(A B C)
> (nreverse lst)
(C B A)
> lst
(A)
```

To reverse `lst`, you have would have to set `lst` to the return value, as with plain `reverse`.

If a function is advertised as destructive, that doesn't mean that it's meant to be called for side-effects. The danger is, some destructive functions give the impression that they are. For example,

```
(nconc x y)
```

almost always has the same effect as

```
(setq x (nconc x y))
```

If you wrote code which relied on the former idiom, it might seem to work for some time. However, it wouldn't do what you expected when `x` was `nil`.

Only a few Lisp operators are intended to be called for side-effects. In general, the built-in operators are meant to be called for their return values. Don't be misled by names like `sort`, `remove`, or `substitute`. If you want side-effects, use `setq` on the return value.

This very rule suggests that some side-effects are inevitable. Having functional programming as an ideal doesn't imply that programs should never have side-effects. It just means that they should have no more than necessary.

It may take time to develop this habit. One way to start is to treat the following operators as if there were a tax on their use:

```
set setq setf psetf psetq incf decf push pop pushnew
rplaca rplacd  rotatef shiftf  remf remprop  remhash
```

and also `let*`, in which imperative programs often lie concealed. Treating these operators as taxable is only proposed as a help toward, not a criterion for, good Lisp style. However, this alone can get you surprisingly far.

In other languages, one of the most common causes of side-effects is the need for a function to return multiple values. If functions can only return one value, they have to "return" the rest by altering their parameters. Fortunately, this isn't necessary in Common Lisp, because any function can return multiple values.

The built-in function `truncate` returns two values, for example—the truncated integer, and what was cut off in order to create it. A typical implementation will print both when `truncate` is called at the toplevel:

```
> (truncate 26.21875)
26
0.21875
```

When the calling code only wants one value, the first one is used:

```
> (= (truncate 26.21875) 26)
T
```

The calling code can catch both return values by using a `multiple-value-bind`. This operator takes a list of variables, a call, and a body of code. The body is evaluated with the variables bound to the respective return values from the call:

```
> (multiple-value-bind (int frac) (truncate 26.21875)
    (list int frac))
(26 0.21875)
```

Finally, to return multiple values, we use the `values` operator:

```
> (defun powers (x)
    (values x (sqrt x) (expt x 2)))
POWERS
> (multiple-value-bind (base root square) (powers 4)
    (list base root square))
(4 2.0 16)
```

Functional programming is a good idea in general. It is a particularly good idea in Lisp, because Lisp has evolved to support it. Built-in operators like `reverse` and `nreverse` are meant to be used in this way. Other operators, like `values` and `multiple-value-bind`, have been provided specifically to make functional programming easier.

3.2 Imperative Outside-In

The aims of functional programming may show more clearly when contrasted with those of the more common approach, imperative programming. A functional program tells you what it wants; an imperative program tells you what to do. A functional program says "Return a list of `a` and the square of the first element of `x`:"

```
(defun fun (x)
  (list 'a (expt (car x) 2)))
```

An imperative programs says "Get the first element of `x`, then square it, then return a list of `a` and the square:"

```
(defun imp (x)
  (let (y sqr)
    (setq y (car x))
    (setq sqr (expt y 2))
    (list 'a sqr)))
```

Lisp users are fortunate in being able to write this program both ways. Some languages are only suited to imperative programming—notably Basic, along with most machine languages. In fact, the definition of `imp` is similar in form to the machine language code that most Lisp compilers would generate for `fun`.

Why write such code when the compiler could do it for you? For many programmers, this question does not even arise. A language stamps its pattern on our thoughts: someone used to programming in an imperative language may have begun to conceive of programs in imperative terms, and may actually find it easier to write imperative programs than functional ones. This habit of mind is worth overcoming if you have a language that will let you.

For alumni of other languages, beginning to use Lisp may be like stepping onto a skating rink for the first time. It's actually much easier to get around on ice than it is on dry land—*if* you use skates. Till then you will be left wondering what people see in this sport.

What skates are to ice, functional programming is to Lisp. Together the two allow you to travel more gracefully, with less effort. But if you are accustomed to another mode of travel, this may not be your experience at first. One of the obstacles to learning Lisp as a second language is learning to program in a functional style.

Fortunately there is a trick for transforming imperative programs into functional ones. You can begin by applying this trick to finished code. Soon you will begin to anticipate yourself, and transform your code as you write it. Soon after that, you will begin to conceive of programs in functional terms from the start.

The trick is to realize that an imperative program is a functional program turned inside-out. To find the functional program implicit in our imperative one, we just turn it outside-in. Let's try this technique on `imp`.

The first thing we notice is the creation of `y` and `sqr` in the initial `let`. This is a sign that bad things are to follow. Like `eval` at runtime, uninitialized variables are so rarely needed that they should generally be treated as a symptom of some illness in the program. Such variables are often used like pins which hold the program down and keep it from coiling into its natural shape.

However, we ignore them for the time being, and go straight to the end of the function. What occurs last in an imperative program occurs outermost in a functional one. So our first step is to grab the final call to `list` and begin stuffing the rest of the program inside it—just like turning a shirt inside-out. We continue by applying the same transformation repeatedly, just as we would with the sleeves of the shirt, and in turn with their cuffs.

Starting at the end, we replace `sqr` with `(expt y 2)`, yielding:

```
(list 'a (expt y 2)))
```

Then we replace `y` by `(car x)`:

```
(list 'a (expt (car x) 2))
```

Now we can throw away the rest of the code, having stuffed it all into the last expression. In the process we removed the need for the variables `y` and `sqr`, so we can discard the `let` as well.

The final result is shorter than what we began with, and easier to understand. In the original code, we're faced with the final expression `(list 'a sqr)`, and it's not immediately clear where the value of `sqr` comes from. Now the source of the return value is laid out for us like a road map.

The example in this section was a short one, but the technique scales up. Indeed, it becomes more valuable as it is applied to larger functions. Even functions which perform side-effects can be cleaned up in the portions which don't.

3.3 Functional Interfaces

Some side-effects are worse than others. For example, though this function calls `nconc`

```
(defun qualify (expr)
  (nconc (copy-list expr) (list 'maybe)))
```

it preserves referential transparency.[2] If you call it with a given argument, it will always return the same (`equal`) value. From the caller's point of view, `qualify` might as well be purely functional code. We can't say the same for `bad-reverse` (page 29), which actually modifies its argument.

Instead of treating all side-effects as equally bad, it would be helpful if we had some way of distinguishing between such cases. Informally, we could say that it's harmless for a function to modify something that no one else owns. For example, the `nconc` in `qualify` is harmless because the list given as the first argument is freshly consed. No one else could own it.

In the general case, we have to talk about ownership not by functions, but by invocations of functions. Though no one else owns the variable `x` here,

```
(let ((x 0))
  (defun total (y)
    (incf x y)))
```

the effects of one call will be visible in succeeding ones. So the rule should be: a given invocation can safely modify what it uniquely owns.

Who owns arguments and return values? The convention in Lisp seems to be that an invocation owns objects it receives as return values, but not objects passed to it as arguments. Functions that modify their arguments are distinguished by the label "destructive," but there is no special name for functions that modify objects returned to them.

This function adheres to the convention, for example:

```
(defun ok (x)
  (nconc (list 'a x) (list 'c)))
```

It calls `nconc`, which doesn't, but since the list spliced by `nconc` will always be freshly made rather than, say, a list passed to `ok` as an argument, `ok` itself is ok.

If it were written slightly differently, however,

```
(defun not-ok (x)
  (nconc (list 'a) x (list 'c)))
```

[2] A definition of referential transparency appears on page 198.

then the call to nconc would be modifying an argument passed to not-ok.

Many Lisp programs violate this convention, at least locally. However, as we saw with ok, local violations need not disqualify the calling function. And functions which do meet the preceding conditions will retain many of the advantages of purely functional code.

To write programs that are really indistinguishable from functional code, we have to add one more condition. Functions can't share objects with other code that doesn't follow the rules. For example, though this function doesn't have side-effects,

```
(defun anything (x)
  (+ x *anything*))
```

its return value depends on the global variable *anything*. So if any other function can alter the value of this variable, anything could return anything.

Code written so that each invocation only modifies what it owns is almost as good as purely functional code. A function that meets all the preceding conditions at least presents a functional interface to the world: if you call it twice with the same arguments, you should get the same results. And this, as the next section will show, is a crucial ingredient in bottom-up programming.

One problem with destructive operations is that, like global variables, they can destroy the locality of a program. When you're writing functional code, you can narrow your focus: you only need consider the functions that call, or are called by, the one you're writing. This benefit disappears when you want to modify something destructively. It could be used anywhere.

The conditions above do not guarantee the perfect locality you get with purely functional code, though they do improve things somewhat. For example, suppose that f calls g as below:

```
(defun f (x)
  (let ((val (g x)))
    ; safe to modify val here?
    ))
```

Is it safe for f to nconc something onto val? Not if g is identity: then we would be modifying something originally passed as an argument to f itself.

So even in programs which do follow the convention, we may have to look beyond f if we want to modify something there. However, we don't have to look as far: instead of worrying about the whole program, we now only have to consider the subtree beginning with f.

A corollary of the convention above is that functions shouldn't return anything that isn't safe to modify. Thus one should avoid writing functions whose return values incorporate quoted objects. If we define `exclaim` so that its return value incorporates a quoted list,

```
(defun exclaim (expression)
  (append expression '(oh my)))
```

Then any later destructive modification of the return value

```
> (exclaim '(lions and tigers and bears))
(LIONS AND TIGERS AND BEARS OH MY)
> (nconc * '(goodness))
(LIONS AND TIGERS AND BEARS OH MY GOODNESS)
```

could alter the list within the function:

```
> (exclaim '(fixnums and bignums and floats))
(FIXNUMS AND BIGNUMS AND FLOATS OH MY GOODNESS)
```

To make `exclaim` proof against such problems, it should be written:

```
(defun exclaim (expression)
  (append expression (list 'oh 'my)))
```

There is one major exception to the rule that functions shouldn't return quoted lists: the functions which generate macro expansions. Macro expanders can safely incorporate quoted lists in the expansions they generate, if the expansions are going straight to the compiler.

Otherwise, one might as well be a suspicious of quoted lists generally. Many other uses of them are likely to be something which ought to be done with a macro like `in` (page 152).

3.4 Interactive Programming

The previous sections presented the functional style as a good way of organizing programs. But it is more than this. Lisp programmers did not adopt the functional style purely for aesthetic reasons. They use it because it makes their work easier. In Lisp's dynamic environment, functional programs can be written with unusual speed, and at the same time, can be unusually reliable.

In Lisp it is comparatively easy to debug programs. A lot of information is available at runtime, which helps in tracing the causes of errors. But even more important is the ease with which you can *test* programs. You don't have to compile a program and test the whole thing at once. You can test functions individually by calling them from the toplevel loop.

Incremental testing is so valuable that Lisp style has evolved to take advantage of it. Programs written in the functional style can be understood one function at a time, and from the point of view of the reader this is its main advantage. However, the functional style is also perfectly adapted to incremental testing: programs written in this style can also be *tested* one function at a time. When a function neither examines nor alters external state, any bugs will appear immediately. Such a function can affect the outside world only through its return values. Insofar as these are what you expected, you can trust the code which produced them.

Experienced Lisp programmers actually design their programs to be easy to test:

1. They try to segregate side-effects in a few functions, allowing the greater part of the program to be written in a purely functional style.
2. If a function must perform side-effects, they try at least to give it a functional interface.
3. They give each function a single, well-defined purpose.

When a function is written, they can test it on a selection of representative cases, then move on to the next one. If each brick does what it's supposed to do, the wall will stand.

In Lisp, the wall can be better-designed as well. Imagine the kind of conversation you would have with someone so far away that there was a transmission delay of one minute. Now imagine speaking to someone in the next room. You wouldn't just have the same conversation faster, you would have a different kind of conversation. In Lisp, developing software is like speaking face-to-face. You can test code as you're writing it. And instant turnaround has just as dramatic an effect on development as it does on conversation. You don't just write the same program faster; you write a different kind of program.

How so? When testing is quicker you can do it more often. In Lisp, as in any language, development is a cycle of writing and testing. But in Lisp the cycle is very short: single functions, or even parts of functions. And if you test everything as you write it, you will know where to look when errors occur: in what you wrote last. Simple as it sounds, this principle is to a large extent what makes bottom-up programming feasible. It brings an extra degree of confidence which enables Lisp programmers to break free, at least part of the time, from the old plan-and-implement style of software development.

Section 1.1 stressed that bottom-up design is an evolutionary process. You build up a language as you write a program in it. This approach can work only if you *trust* the lower levels of code. If you really want to use this layer as a language, you have to be able to assume, as you would with any language, that any bugs you encounter are bugs in your application and not in the language itself.

So your new abstractions are supposed to bear this heavy burden of responsibility, and yet you're supposed to just spin them off as the need arises? Just so; in Lisp you can have both. When you write programs in a functional style and test them incrementally, you can have the flexibility of doing things on the spur of the moment, plus the kind of reliability one usually associates with careful planning.

4

Utility Functions

Common Lisp operators come in three types: functions and macros, which you can write yourself, and special forms, which you can't. This chapter describes techniques for extending Lisp with new functions. But "techniques" here means something different from what it usually does. The important thing to know about such functions is not how they're written, but where they come from. An extension to Lisp will be written using mostly the same techniques you would use to write any other Lisp function. The hard part of writing these extensions is not deciding how to write them, but deciding which ones to write.

4.1 Birth of a Utility

In its simplest form, bottom-up programming means second-guessing whoever designed your Lisp. At the same time as you write your program, you also add to Lisp new operators which make your program easy to write. These new operators are called *utilities.*

The term "utility" has no precise definition. A piece of code can be called a utility if it seems too small to be considered as a separate application, and too general-purpose to be considered as part of a particular program. A database program would not be a utility, for example, but a function which performed a single operation on a list could be. Most utilities resemble the functions and macros that Lisp has already. In fact, many of Common Lisp's built-in operators began life as utilities. The function `remove-if-not`, which collects all the elements of a list satisfying some predicate, was defined by individual programmers for years before it became a part of Common Lisp.

Learning to write utilities would be better described as learning the habit of writing them, rather than the technique of writing them. Bottom-up programming means simultaneously writing a program and a programming language. To do this well, you have to develop a fine sense of which operators a program is lacking. You have to be able to look at a program and say, "Ah, what you really mean to say is *this*."

For example, suppose that `nicknames` is a function which takes a name and builds a list of all the nicknames which could be derived from it. Given this function, how do we collect all the nicknames yielded by a list of names? Someone learning Lisp might write a function like:

```
(defun all-nicknames (names)
  (if (null names)
      nil
      (nconc (nicknames (car names))
             (all-nicknames (cdr names)))))
```

A more experienced Lisp programmer can look at such a function and say "Ah, what you really want is `mapcan`." Then instead of having to define and call a new function to find all the nicknames of a group of people, you can use a single expression:

```
(mapcan #'nicknames people)
```

The definition of `all-nicknames` is reinventing the wheel. However, that's not all that's wrong with it: it is also burying in a specific function something that could be done by a general-purpose operator.

In this case the operator, `mapcan`, already exists. Anyone who knew about `mapcan` would feel a little uncomfortable looking at `all-nicknames`. To be good at bottom-up programming is to feel equally uncomfortable when the missing operator is one which hasn't been written yet. You must be able to say "what you really want is `x`," and at the same time, to know what `x` should be.

Lisp programming entails, among other things, spinning off new utilities as you need them. The aim of this section is to show how such utilities are born. Suppose that `towns` is a list of nearby towns, sorted from nearest to farthest, and that `bookshops` is a function which returns a list of all the bookshops in a city. If we want to find the nearest town which has any bookshops, and the bookshops in it, we could begin with:

```
(let ((town (find-if #'bookshops towns)))
  (values town (bookshops town)))
```

But this is a bit inelegant: when `find-if` finds an element for which `bookshops` returns a non-nil value, the value is thrown away, only to be recomputed as soon as `find-if` returns. If `bookshops` were an expensive call, this idiom would be inefficient as well as ugly. To avoid unnecessary work, we could use the following function instead:

```
(defun find-books (towns)
  (if (null towns)
      nil
      (let ((shops (bookshops (car towns))))
        (if shops
            (values (car towns) shops)
            (find-books (cdr towns))))))
```

Then calling `(find-books towns)` would at least get us what we wanted with no more computation than necessary. But wait—isn't it likely that at some time in the future we will want to do the same kind of search again? What we really want here is a utility which combines `find-if` and `some`, returning both the successful element, and the value returned by the test function. Such a utility could be defined as:

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
        (if val
            (values (car lst) val)
            (find2 fn (cdr lst))))))
```

Notice the similarity between `find-books` and `find2`. Indeed, the latter could be described as the skeleton of the former. Now, using the new utility, we can achieve our original aim with a single expression:

```
(find2 #'bookshops towns)
```

One of the unique characteristics of Lisp programming is the important role of functions as arguments. This is part of why Lisp is well-adapted to bottom-up programming. It's easier to abstract out the bones of a function when you can pass the flesh back as a functional argument.

Introductory programming courses teach early on that abstraction leads to less duplication of effort. One of the first lessons is: don't wire in behavior. For example, instead of defining two functions which do the same thing but for one or two constants, define a single function and pass the constants as arguments.

In Lisp we can carry this idea further, because we can pass whole functions as arguments. In both of the previous examples we went from a specific function to a more general function which took a functional argument. In the first case we used the predefined `mapcan` and in the second we wrote a new utility, `find2`, but the general principle is the same: instead of mixing the general and the specific, define the general and pass the specific as an argument.

When carefully applied, this principle yields noticeably more elegant programs. It is not the only force driving bottom-up design, but it is a major one. Of the 32 utilities defined in this chapter, 18 take functional arguments.

4.2 Invest in Abstraction

If brevity is the soul of wit, it is also, along with efficiency, the essence of good software. The cost of writing or maintaining a program increases with its length. ○ All other things being equal, the shorter program is the better.

From this point of view, the writing of utilities should be treated as a capital expenditure. By replacing `find-books` with the utility `find2`, we end up with just as many lines of code. But we have made the program shorter in one sense, because the length of the utility does not have to be charged against the current program.

It is not just an accounting trick to treat extensions to Lisp as capital expenditures. Utilities can go into a separate file; they will not clutter our view as we're working on the program, nor are they likely to be involved if we have to return later to change the program in some respect.

As capital expenditures, however, utilities demand extra attention. It is especially important that they be well-written. They are going to be used repeatedly, so any incorrectness or inefficiency will be multiplied. Extra care must also go into their design: a new utility must be written for the general case, not just for the problem at hand. Finally, like any capital expenditure, we need not be in a hurry about it. If you're thinking of spinning off some new operator, but aren't sure that you will want it elsewhere, write it anyway, but leave it with the particular program which uses it. Later if you use the new operator in other programs, you can promote it from a subroutine to a utility and make it generally accessible.

The utility `find2` seems to be a good investment. By making a capital outlay of 7 lines, we get an immediate savings of 7. The utility has paid for itself in the first use. A programming language, Guy Steele wrote, should "cooperate with our natural tendency towards brevity:"

> ...we tend to believe that the expense of a programming construct is proportional to the amount of writer's cramp that it causes us (by "belief" I mean here an unconscious tendency rather than a fervent

> conviction). Indeed, this is not a bad psychological principle for language designers to keep in mind. We think of addition as cheap partly because we can notate it with a single character: "+". Even if we believe that a construct is expensive, we will often prefer it to a cheaper one if it will cut our writing effort in half.

In any language, the "tendency towards brevity" will cause trouble unless it is allowed to vent itself in new utilities. The shortest idioms are rarely the most efficient ones. If we want to know whether one list is longer than another, raw Lisp will tempt us to write

```
(> (length x) (length y))
```

If we want to map a function over several lists, we will likewise be tempted to join them together first:

```
(mapcar fn (append x y z))
```

Such examples show that it's especially important to write utilities for situations we might otherwise handle inefficiently. A language augmented with the right utilities will lead us to write more abstract programs. If these utilities are properly defined, it will also lead us to write more efficient ones.

A collection of utilities will certainly make programming easier. But they can do more than that: they can make you write better programs. The muses, like cooks, spring into action at the sight of ingredients. This is why artists like to have a lot of tools and materials in their studios. They know that they are more likely to start something new if they have what they need ready at hand. The same phenomenon appears with programs written bottom-up. Once you have written a new utility, you may find yourself using it more than you would have expected.

The following sections describe several classes of utility functions. They do not by any means represent all the different types of functions you might add to Lisp. However, all the utilities given as examples are ones that have proven their worth in practice.

4.3 Operations on Lists

Lists were originally Lisp's main data structure. Indeed, the name "Lisp" comes from "LISt Processing." It is as well not to be misled by this historical fact, however. Lisp is not inherently about processing lists any more than Polo shirts are for Polo. A highly optimized Common Lisp program might never see a list.

It would still *be* a list, though, at least at compile-time. The most sophisticated programs, which use lists less at runtime, use them proportionately more at

```
(proclaim '(inline last1 single append1 conc1 mklist))

(defun last1 (lst)
  (car (last lst)))

(defun single (lst)
  (and (consp lst) (not (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun conc1 (lst obj)
  (nconc lst (list obj)))

(defun mklist (obj)
  (if (listp obj) obj (list obj)))
```

Figure 4.1: Small functions which operate on lists.

compile-time, when generating macro expansions. So although the role of lists is decreased in modern dialects, operations on lists can still make up the greater part of a Lisp program.

Figures 4.1 and 4.2 contain a selection of functions which build or examine lists. Those given in Figure 4.1 are among the smallest utilities worth defining. For efficiency, they should all be declared inline (page 26).

The first, `last1`, returns the last element in a list. The built-in function `last` returns the last *cons* in a list, not the last element. Most of the time one uses it to get the last element, by saying `(car (last ...))`. Is it worth writing a new utility for such a case? Yes, when it effectively replaces one of the built-in operators.

Notice that `last1` does no error-checking. In general, none of the code defined in this book will do error-checking. Partly this is just to make the examples clearer. But in shorter utilities it is reasonable not to do any error-checking anyway. If we try:

```
> (last1 "blub")
>>Error: "blub" is not a list.
Broken at LAST...
```

the error will be caught by `last` itself. When utilities are small, they form a layer of abstraction so thin that it starts to be transparent. As one can see through a thin layer of ice, one can see through utilities like `last1` to interpret errors which arise in the underlying functions.

The function `single` tests whether something is a list of one element. Lisp programs need to make this test rather often. At first one might be tempted to use the natural translation from English:

```
(= (length lst) 1)
```

Written this way, the test would be very inefficient. We know all we need to know as soon as we've looked past the first element.

Next come `append1` and `conc1`. Both attach a new element to the end of a list, the latter destructively. These functions are small, but so frequently needed that they are worth defining. Indeed, `append1` has been predefined in previous Lisp dialects.

So has `mklist`, which was predefined in (at least) Interlisp. Its purpose is to ensure that something is a list. Many Lisp functions are written to return either a single value or a list of values. Suppose that `lookup` is such a function, and that we want to collect the results of calling it on all the elements of a list called `data`. We can do so by writing:

```
(mapcan #'(lambda (d) (mklist (lookup d)))
        data)
```

Figure 4.2 contains some larger examples of list utilities. The first, `longer`, is useful from the point of view of efficiency as well as abstraction. It compares two sequences and returns true only if the first is longer. When comparing the lengths of two lists, it is tempting to do just that:

```
(> (length x) (length y))
```

This idiom is inefficient because it requires the program to traverse the entire length of both lists. If one list is much longer than the other, all the effort of traversing the difference in their lengths will be wasted. It is faster to do as `longer` does and traverse the two lists in parallel.

Embedded within `longer` is a recursive function to compare the lengths of two lists. Since `longer` is for comparing lengths, it should work for anything that you could give as an argument to `length`. But the possibility of comparing lengths in parallel only applies to lists, so the internal function is only called if both arguments are lists.

The next function, `filter`, is to `some` what `remove-if-not` is to `find-if`. The built-in `remove-if-not` returns all the values that might have been returned if you called `find-if` with the same function on successive cdrs of a list. Analogously, `filter` returns what `some` would have returned for successive cdrs of the list:

```
(defun longer (x y)
  (labels ((compare (x y)
             (and (consp x)
                  (or (null y)
                      (compare (cdr x) (cdr y))))))
    (if (and (listp x) (listp y))
        (compare x y)
        (> (length x) (length y)))))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
             (let ((rest (nthcdr n source)))
               (if (consp rest)
                   (rec rest (cons (subseq source 0 n) acc))
                   (nreverse (cons source acc))))))
    (if source (rec source nil) nil)))
```

Figure 4.2: Larger functions that operate on lists.

```
> (filter #'(lambda (x) (if (numberp x) (1+ x)))
          '(a 1 2 b 3 c d 4))
(2 3 4 5)
```

You give `filter` a function and a list, and get back a list of whatever non-nil values are returned by the function as it is applied to the elements of the list.

Notice that `filter` uses an accumulator in the same way as the tail-recursive functions described in Section 2.8. Indeed, the aim in writing a tail-recursive function is to have the compiler generate code in the shape of `filter`. For `filter`, the straightforward iterative definition is simpler than the tail-recursive one. The combination of `push` and `nreverse` in the definition of `filter` is the standard Lisp idiom for accumulating a list.

The last function in Figure 4.2 is for grouping lists into sublists. You give `group` a list *l* and a number *n*, and it will return a new list in which the elements of l are grouped into sublists of length n. The remainder is put in a final sublist. Thus if we give 2 as the second argument, we get an assoc-list:

```
> (group '(a b c d e f g) 2)
((A B) (C D) (E F) (G))
```

This function is written in a rather convoluted way in order to make it tail-recursive (Section 2.8). The principle of rapid prototyping applies to individual functions as well as to whole programs. When writing a function like `flatten`, it can be a good idea to begin with the simplest possible implementation. Then, once the simpler version works, you can replace it if necessary with a more efficient tail-recursive or iterative version. If it's short enough, the initial version could be left as a comment to describe the behavior of its replacement. (Simpler versions of `group` and several other functions in Figures 4.2 and 4.3 are included in the ○ note on page 389.)

The definition of `group` is unusual in that it checks for at least one error: a second argument of `0`, which would otherwise send the function into an infinite recursion.

In one respect, the examples in this book deviate from usual Lisp practice: to make the chapters independent of one another, the code examples are as much as possible written in raw Lisp. Because it is so useful in defining macros, `group` is an exception, and will reappear at several points in later chapters.

The functions in Figure 4.2 all work their way along the top-level structure of a list. Figure 4.3 shows two examples of functions that descend into nested lists. The first, `flatten`, was also predefined in Interlisp. It returns a list of all the atoms that are elements of a list, or elements of its elements, and so on:

```
> (flatten '(a (b c) ((d e) f)))
(A B C D E F)
```

The other function in Figure 4.3, `prune`, is to `remove-if` as `copy-tree` is to `copy-list`. That is, it recurses down into sublists:

```
> (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
(1 (3 (5)) 7 (9))
```

Every leaf for which the function returns true is removed.

4.4 Search

This section gives some examples of functions for searching lists. Common Lisp provides a rich set of built-in operators for this purpose, but some tasks

```
(defun flatten (x)
  (labels ((rec (x acc)
             (cond ((null x) acc)
                   ((atom x) (cons x acc))
                   (t (rec (car x) (rec (cdr x) acc))))))
    (rec x nil)))

(defun prune (test tree)
  (labels ((rec (tree acc)
             (cond ((null tree) (nreverse acc))
                   ((consp (car tree))
                    (rec (cdr tree)
                         (cons (rec (car tree) nil) acc)))
                   (t (rec (cdr tree)
                           (if (funcall test (car tree))
                               acc
                               (cons (car tree) acc)))))))
    (rec tree nil)))
```

Figure 4.3: Doubly-recursive list utilities.

are still difficult—or at least difficult to perform efficiently. We saw this in the
○ hypothetical case described on page 41. The first utility in Figure 4.4, `find2`, is the one we defined in response to it.

The next utility, `before`, is written with similar intentions. It tells you if one object is found before another in a list:

```
> (before 'b 'd '(a b c d))
(B C D)
```

It is easy enough to do this sloppily in raw Lisp:

```
(< (position 'b '(a b c d)) (position 'd '(a b c d)))
```

But the latter idiom is inefficient and error-prone: inefficient because we don't need to find both objects, only the one that occurs first; and error-prone because if either object isn't in the list, `nil` will be passed as an argument to `<`. Using `before` fixes both problems.

Since `before` is similar in spirit to a test for membership, it is written to resemble the built-in `member` function. Like `member` it takes an optional `test` argument, which defaults to `eql`. Also, instead of simply returning `t`, it tries to

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
        (if val
            (values (car lst) val)
            (find2 fn (cdr lst))))))

(defun before (x y lst &key (test #'eql))
  (and lst
       (let ((first (car lst)))
         (cond ((funcall test y first) nil)
               ((funcall test x first) lst)
               (t (before x y (cdr lst) :test test))))))

(defun after (x y lst &key (test #'eql))
  (let ((rest (before y x lst :test test)))
    (and rest (member x rest :test test))))

(defun duplicate (obj lst &key (test #'eql))
  (member obj (cdr (member obj lst :test test))
          :test test))

(defun split-if (fn lst)
  (let ((acc nil))
    (do ((src lst (cdr src)))
        ((or (null src) (funcall fn (car src)))
         (values (nreverse acc) src))
      (push (car src) acc))))
```

Figure 4.4: Functions which search lists.

return potentially useful information: the cdr beginning with the object given as the first argument.

Note that `before` returns true if we encounter the first argument before encountering the second. Thus it will return true if the second argument doesn't occur in the list at all:

```
> (before 'a 'b '(a))
(A)
```

We can peform a more exacting test by calling `after`, which requires that both its arguments occur in the list:

```
> (after 'a 'b '(b a d))
(A D)
> (after 'a 'b '(a))
NIL
```

If (`member` *o l*) finds *o* in the list *l,* it also returns the cdr of *l* beginning with *o*. This return value can be used, for example, to test for duplication. If *o* is duplicated in *l,* then it will also be found in the cdr of the list returned by `member`. This idiom is embodied in the next utility, `duplicate`:

```
> (duplicate 'a '(a b c a d))
(A D)
```

Other utilities to test for duplication could be written on the same principle.

More fastidious language designers are shocked that Common Lisp uses `nil` to represent both falsity and the empty list. It does cause trouble sometimes (see Section 14.2), but it is convenient in functions like `duplicate`. In questions of sequence membership, it seems natural to represent falsity as the empty sequence.

The last function in Figure 4.4 is also a kind of generalization of `member`. While `member` returns the cdr of the list beginning with the element it finds, `split-if` returns both halves. This utility is mainly used with lists that are ordered in some respect:

```
> (split-if #'(lambda (x) (> x 4))
            '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4)
(5 6 7 8 9 10)
```

Figure 4.5 contains search functions of another kind: those which compare elements against one another. The first, `most`, looks at one element at a time. It takes a list and a scoring function, and returns the element with the highest score. In case of ties, the element occurring first wins.

```
> (most #'length '((a b) (a b c) (a) (e f g)))
(A B C)
3
```

For convenience, `most` also returns the score of the winner.

A more general kind of search is provided by `best`. This utility also takes a function and a list, but here the function must be a predicate of two arguments. It returns the element which, according to the predicate, beats all the others.

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setq wins obj
                    max  score))))
        (values wins max))))

(defun best (fn lst)
  (if (null lst)
      nil
      (let ((wins (car lst)))
        (dolist (obj (cdr lst))
          (if (funcall fn obj wins)
              (setq wins obj)))
        wins)))

(defun mostn (fn lst)
  (if (null lst)
      (values nil nil)
      (let ((result (list (car lst)))
            (max (funcall fn (car lst))))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (cond ((> score max)
                   (setq max    score
                         result (list obj)))
                  ((= score max)
                   (push obj result)))))
        (values (nreverse result) max))))
```

Figure 4.5: Search functions which compare elements.

```
> (best #'> '(1 2 3 4 5))
5
```

We can think of `best` as being equivalent to `car` of `sort`, but much more efficient.

It is up to the caller to provide a predicate which defines a total order on the elements of the list. Otherwise the order of the elements will influence the result; as before, in case of ties, the first element wins.

Finally, `mostn` takes a function and a list and returns a *list* of all the elements for which the function yields the highest score (along with the score itself):

```
> (mostn  #'length '((a b) (a b c) (a) (e f g)))
((A B C) (E F G))
3
```

4.5 Mapping

Another widely used class of Lisp functions are the mapping functions, which apply a function to a sequence of arguments. Figure 4.6 shows some examples of new mapping functions. The first three are for applying a function to a range of numbers without having to cons up a list to contain them. The first two, `map0-n` and `map1-n`, work for ranges of positive integers:

```
> (map0-n #'1+ 5)
(1 2 3 4 5 6)
```

Both are written using the more general `mapa-b`, which works for any range of numbers:

```
> (mapa-b #'1+ -2 0 .5)
(-1 -0.5 0.0 0.5 1.0)
```

Following `mapa-b` is the still more general `map->`, which works for sequences of objects of any kind. The sequence begins with the object given as the second argument, the end of the sequence is defined by the function given as the third argument, and successors are generated by the function given as the fourth argument. With `map->` it is possible to navigate arbitrary data structures, as well as operate on sequences of numbers. We could define `mapa-b` in terms of `map->` as follows:

```
(defun mapa-b (fn a b &optional (step 1))
  (map-> fn
         a
         #'(lambda (x) (> x b))
         #'(lambda (x) (+ x step))))
```

```
(defun map0-n (fn n)
  (mapa-b fn 0 n))

(defun map1-n (fn n)
  (mapa-b fn 1 n))

(defun mapa-b (fn a b &optional (step 1))
  (do ((i a (+ i step))
       (result nil))
      ((> i b) (nreverse result))
    (push (funcall fn i) result)))

(defun map-> (fn start test-fn succ-fn)
  (do ((i start (funcall succ-fn i))
       (result nil))
      ((funcall test-fn i) (nreverse result))
    (push (funcall fn i) result)))

(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))

(defun mapcars (fn &rest lsts)
  (let ((result nil))
    (dolist (lst lsts)
      (dolist (obj lst)
        (push (funcall fn obj) result)))
    (nreverse result)))

(defun rmapcar (fn &rest args)
  (if (some #'atom args)
      (apply fn args)
      (apply #'mapcar
             #'(lambda (&rest args)
                 (apply #'rmapcar fn args))
             args)))
```

Figure 4.6: Mapping functions.

For efficiency, the built-in mapcan is destructive. It could be duplicated by:

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

Because mapcan splices together lists with nconc, the lists returned by the first argument had better be newly created, or the next time we look at them they might be altered. That's why nicknames (page 41) was defined as a function which "builds a list" of nicknames. If it simply returned a list stored elsewhere, it wouldn't have been safe to use mapcan. Instead we would have had to splice the returned lists with append. For such cases, mappend offers a nondestructive alternative to mapcan.

The next utility, mapcars, is for cases where we want to mapcar a function over several lists. If we have two lists of numbers and we want to get a single list of the square roots of both, using raw Lisp we could say

```
(mapcar #'sqrt (append list1 list2))
```

but this conses unnecessarily. We append together list1 and list2 only to discard the result immediately. With mapcars we can get the same result from:

```
(mapcars #'sqrt list1 list2)
```

and do no unnecessary consing.

The final function in Figure 4.6 is a version of mapcar for trees. Its name, rmapcar, is short for "recursive mapcar," and what mapcar does on flat lists, it does on trees:

```
> (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
123456789
(1 2 (3 4 (5) 6) 7 (8 9))
```

Like mapcar, it can take more than one list argument.

```
> (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
(11 (22 (33) 44))
```

Several of the functions which appear later on ought really to call rmapcar, including rep_ on page 324.

To some extent, traditional list mapping functions may be rendered obsolete by the new series macros introduced in CLTL2. For example,

```
(mapa-b #'fn a b c)
```

could be rendered

```
(defun readlist (&rest args)
  (values (read-from-string
            (concatenate 'string "("
                                 (apply #'read-line args)
                                 ")"))))

(defun prompt (&rest args)
  (apply #'format *query-io* args)
  (read *query-io*))

(defun break-loop (fn quit &rest args)
  (format *query-io* "Entering break-loop.~%")
  (loop
    (let ((in (apply #'prompt args)))
      (if (funcall quit in)
          (return)
          (format *query-io* "~A~%" (funcall fn in))))))
```

Figure 4.7: I/O functions.

```
(collect (#Mfn (scan-range :from a :upto b :by c)))
```

However, there is still some call for mapping functions. A mapping function may in some cases be clearer or more elegant. Some things we could express with `map->` might be difficult to express using series. Finally, mapping functions, as functions, can be passed as arguments.

4.6 I/O

Figure 4.7 contains three examples of I/O utilities. The need for this kind of utility varies from program to program. Those in Figure 4.7 are just a representative sample. The first is for the case where you want users to be able to type in expressions without parentheses; it reads a line of input and returns it as a list:

```
> (readlist)
Call me "Ed"
(CALL ME "Ed")
```

The call to `values` ensures that we get only one value back (`read-from-string` itself returns a second value that is irrelevant in this case).

The function `prompt` combines printing a question and reading the answer. It takes the arguments of `format`, except the initial stream argument.

```
> (prompt "Enter a number between ~A and ~A.~%>> " 1 10)
Enter a number between 1 and 10.
>> 3
3
```

Finally, `break-loop` is for situations where you want to imitate the Lisp toplevel. It takes two functions and an `&rest` argument, which is repeatedly given to `prompt`. As long as the second function returns false for the input, the first function is applied to it. So for example we could simulate the actual Lisp toplevel with:

```
> (break-loop #'eval #'(lambda (x) (eq x :q)) ">> ")
Entering break-loop.
>> (+ 2 3)
5
>> :q
:Q
```

This, by the way, is the reason Common Lisp vendors generally insist on runtime licenses. If you can call `eval` at runtime, then any Lisp program can include Lisp.

4.7 Symbols and Strings

Symbols and strings are closely related. By means of printing and reading functions we can go back and forth between the two representations. Figure 4.8 contains examples of utilities which operate on this border. The first, `mkstr`, takes any number of arguments and concatenates their printed representations into a string:

```
> (mkstr pi " pieces of " 'pi)
"3.141592653589793 pieces of PI"
```

Built upon it is `symb`, which is mostly used for building symbols. It takes one or more arguments and returns the symbol (creating one if necessary) whose print-name is their concatenation. It can take as an argument any object which has a printable representation: symbols, strings, numbers, even lists.

```
> (symb 'ar "Madi" #\L #\L 0)
|ARMadiLL0|
```

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
    (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))

(defun reread (&rest args)
  (values (read-from-string (apply #'mkstr args))))

(defun explode (sym)
  (map 'list #'(lambda (c)
                 (intern (make-string 1
                                      :initial-element c)))
             (symbol-name sym)))
```

Figure 4.8: Functions which operate on symbols and strings.

After calling `mkstr` to concatenate all its arguments into a single string, `symb` sends the string to `intern`. This function is Lisp's traditional symbol-builder: it takes a string and either finds the symbol which prints as the string, or makes a new one which does.

Any string can be the print-name of a symbol, even a string containing lower-case letters or macro characters like parentheses. When a symbol's name contains such oddities, it is printed within vertical bars, as above. In source code, such symbols should either be enclosed in vertical bars, or the offending characters preceded by backslashes:

```
> (let ((s (symb '(a b))))
    (and (eq s '|(A B)|) (eq s '\(A\ B\))))
T
```

The next function, `reread`, is a generalization of `symb`. It takes a series of objects, and prints and rereads them. It can return symbols like `symb`, but it can also return anything else that `read` can. Read-macros will be invoked instead of being treated as part of a symbol's name, and `a:b` will be read as the symbol `b` in package `a`, instead of the symbol `|a:b|` in the current package.[1] The more general function is also pickier: `reread` will generate an error if its arguments are not proper Lisp syntax.

[1] For an introduction to packages, see the Appendix beginning on page 381.

The last function in Figure 4.8 was predefined in several earlier dialects: `explode` takes a symbol and returns a list of symbols made from the characters in its name.

```
> (explode 'bomb)
(B O M B)
```

It is no accident that this function wasn't included in Common Lisp. If you find yourself wanting to take apart symbols, you're probably doing something inefficient. However, there is a place for this kind of utility in prototypes, if not in production software.

4.8 Density

If your code uses a lot of new utilities, some readers may complain that it is hard to understand. People who are not yet very fluent in Lisp will only be used to reading raw Lisp. In fact, they may not be used to the idea of an extensible language at all. When they look at a program which depends heavily on utilities, it may seem to them that the author has, out of pure eccentricity, decided to write the program in some sort of private language.

All these new operators, it might be argued, make the program harder to read. One has to understand them all before being able to read the program. To see why this kind of statement is mistaken, consider the case described on page 41, in which we want to find the nearest bookshops. If you wrote the program using `find2`, someone could complain that they had to understand the definition of this new utility before they could read your program. Well, suppose you hadn't used `find2`. Then, instead of having to understand the definition of `find2`, the reader would have had to understand the definition of `find-books`, in which the function of `find2` is mixed up with the specific task of finding bookshops. It is no more difficult to understand `find2` than `find-books`. And here we have only used the new utility once. Utilities are meant to be used repeatedly. In a real program, it might be a choice between having to understand `find2`, and having to understand three or four specialized search routines. Surely the former is easier.

So yes, reading a bottom-up program requires one to understand all the new operators defined by the author. But this will nearly always be less work than having to understand all the code that would have been required without them.

If people complain that using utilities makes your code hard to read, they probably don't realize what the code would look like if you hadn't used them. Bottom-up programming makes what would otherwise be a large program look like a small, simple one. This can give the impression that the program doesn't do much, and should therefore be easy to read. When inexperienced readers look closer and find that this isn't so, they react with dismay.

We find the same phenomenon in other fields: a well-designed machine may have fewer parts, and yet look more complicated, because it is packed into a smaller space. Bottom-up programs are conceptually denser. It may take an effort to read them, but not as much as it would take if they hadn't been written that way.

There is one case in which you might deliberately avoid using utilities: if you had to write a small program to be distributed independently of the rest of your code. A utility usually pays for itself after two or three uses, but in a small program, a utility might not be used enough to justify including it.

5

Returning Functions

The previous chapter showed how the ability to pass functions as arguments leads to greater possibilities for abstraction. The more we can do to functions, the more we can take advantage of these possibilities. By defining functions to build and return new functions, we can magnify the effect of utilities which take functions as arguments.

The utilities in this chapter operate on functions. It would be more natural, at least in Common Lisp, to write many of them to operate on expressions—that is, as macros. A layer of macros will be superimposed on some of these operators in Chapter 15. However, it is important to know what part of the task can be done with functions, even if we will eventually call these functions only through macros.

5.1 Common Lisp Evolves

Common Lisp originally provided several pairs of complementary functions. The functions `remove-if` and `remove-if-not` make one such pair. If `pred` is a predicate of one argument, then

```
(remove-if-not #'pred lst)
```

is equivalent to

```
(remove-if #'(lambda (x) (not (pred x))) lst)
```

By varying the function given as an argument to one, we can duplicate the effect of the other. In that case, why have both? CLTL2 includes a new function intended for cases like this: `complement` takes a predicate *p* and returns a function which always returns the opposite value. When *p* returns true, the complement returns false, and vice versa. Now we can replace

```
(remove-if-not #'pred lst)
```

with the equivalent

```
(remove-if (complement #'pred) lst)
```

With `complement`, there is little justification for continuing to use the `-if-not` functions.[1] Indeed, CLTL2 (p. 391) says that their use is now deprecated. If they remain in Common Lisp, it will only be for the sake of compatibility.

The new `complement` operator is the tip of an important iceberg: functions which return functions. This has long been an important part of the idiom of Scheme. Scheme was the first Lisp to make functions lexical closures, and it is this which makes it interesting to have functions as return values.

It's not that we couldn't return functions in a dynamically scoped Lisp. The following function would work the same under dynamic or lexical scope:

```
(defun joiner (obj)
  (typecase obj
    (cons   #'append)
    (number #'+)))
```

It takes an object and, depending on its type, returns a function to add such objects together. We could use it to define a polymorphic `join` function that worked for numbers or lists:

```
(defun join (&rest args)
  (apply (joiner (car args)) args))
```

However, returning constant functions is the limit of what we can do with dynamic scope. What we can't do (well) is build functions at runtime; `joiner` can return one of two functions, but the two choices are fixed.

On page 18 we saw another function for returning functions, which relied on lexical scope:

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

[1] Except perhaps `remove-if-not`, which is used more often than `remove-if`.

Calling `make-adder` will yield a closure whose behavior depends on the value originally given as an argument:

```
> (setq add3 (make-adder 3))
#<Interpreted-Function BF1356>
> (funcall add3 2)
5
```

Under lexical scope, instead of merely choosing among a group of constant functions, we can build new closures at runtime. With dynamic scope this technique is impossible.[2] If we consider how `complement` would be written, we see that it too must return a closure:

```
(defun complement (fn)
  #'(lambda (&rest args) (not (apply fn args))))
```

The function returned by `complement` uses the value of the parameter `fn` when `complement` was called. So instead of just choosing from a group of constant functions, `complement` can custom-build the inverse of any function:

```
> (remove-if (complement #'oddp) '(1 2 3 4 5 6))
(1 3 5)
```

Being able to pass functions as arguments is a powerful tool for abstraction. The ability to write functions which return functions allows us to make the most of it. The remaining sections present several examples of utilities which return functions.

5.2 Orthogonality

An *orthogonal* language is one in which you can express a lot by combining a small number of operators in a lot of different ways. Toy blocks are very orthogonal; a plastic model kit is hardly orthogonal at all. The main advantage of `complement` is that it makes a language more orthogonal. Before `complement`, Common Lisp had pairs of functions like `remove-if` and `remove-if-not`, `subst-if` and `subst-if-not`, and so on. With `complement` we can do without half of them.

The `setf` macro also improves Lisp's orthogonality. Earlier dialects of Lisp would often have pairs of functions for reading and writing data. With property-lists, for example, there would be one function to establish properties and another function to ask about them. In Common Lisp, we have only the latter, `get`. To

[2] Under dynamic scope, we could write something like `make-adder`, but it would hardly ever work. The binding of `n` would be determined by the environment in which the returned function was eventually called, and we might not have any control over that.

```
(defvar *!equivs* (make-hash-table))

(defun ! (fn)
  (or (gethash fn *!equivs*) fn))

(defun def! (fn fn!)
  (setf (gethash fn *!equivs*) fn!))
```

Figure 5.1: Returning destructive equivalents.

establish a property, we use `get` in combination with `setf`:

```
(setf (get 'ball 'color) 'red)
```

We may not be able to make Common Lisp smaller, but we can do something almost as good: use a smaller subset of it. Can we define any new operators which would, like `complement` and `setf`, help us toward this goal? There is at least one other way in which functions are grouped in pairs. Many functions also come in a destructive version: `remove-if` and `delete-if`, `reverse` and `nreverse`, `append` and `nconc`. By defining an operator to return the destructive counterpart of a function, we would not have to refer to the destructive functions directly.

Figure 5.1 contains code to support the notion of destructive counterparts. The global hash-table `*!equivs*` maps functions to their destructive equivalents; `!` returns destructive equivalents; and `def!` sets them. The name of the `!` (bang) operator comes from the Scheme convention of appending `!` to the names of functions with side-effects. Now once we have defined

```
(def! #'remove-if #'delete-if)
```

then instead of

```
(delete-if #'oddp lst)
```

we would say

```
(funcall (! #'remove-if) #'oddp lst)
```

Here the awkwardness of Common Lisp masks the basic elegance of the idea, which would be more visible in Scheme:

```
((! remove-if) oddp lst)
```

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal)))
    #'(lambda (&rest args)
        (multiple-value-bind (val win) (gethash args cache)
          (if win
              val
              (setf (gethash args cache)
                    (apply fn args)))))))
```

Figure 5.2: Memoizing utility.

As well as greater orthogonality, the `!` operator brings a couple of other benefits. It makes programs clearer, because we can see immediately that `(! #'foo)` is the destructive equivalent of `foo`. Also, it gives destructive operations a distinct, recognizable form in source code, which is good because they should receive special attention when we are searching for a bug.

Since the relation between a function and its destructive counterpart will usually be known before runtime, it would be most efficient to define `!` as a macro, or even provide a read macro for it.

5.3 Memoizing

If some function is expensive to compute, and we expect sometimes to make the same call more than once, then it pays to *memoize:* to cache the return values of all the previous calls, and each time the function is about to be called, to look first in the cache to see if the value is already known.

Figure 5.2 contains a generalized memoizing utility. We give a function to `memoize`, and it returns an equivalent memoized version—a closure containing a hash-table in which to store the results of previous calls.

```
> (setq slowid (memoize #'(lambda (x) (sleep 5) x)))
#<Interpreted-Function C38346>
> (time (funcall slowid 1))
Elapsed Time = 5.15 seconds
1
> (time (funcall slowid 1))
Elapsed Time = 0.00 seconds
1
```

With a memoized function, repeated calls are just hash-table lookups. There is of course the additional expense of a lookup on each initial call, but since we

```
(defun compose (&rest fns)
  (if fns
      (let ((fn1 (car (last fns)))
            (fns (butlast fns)))
        #'(lambda (&rest args)
            (reduce #'funcall fns
                    :from-end t
                    :initial-value (apply fn1 args))))
      #'identity))
```

Figure 5.3: An operator for functional composition.

would only memoize a function that was sufficiently expensive to compute, it's reasonable to assume that this cost is insignificant in comparison.

Though adequate for most uses, this implementation of `memoize` has several limitations. It treats calls as identical if they have `equal` argument lists; this could be too strict if the function had keyword parameters. Also, it is intended only for single-valued functions, and cannot store or return multiple values.

5.4 Composing Functions

The complement of a function f is denoted $\sim f$. Section 5.1 showed that closures make it possible to define $\sim$ as a Lisp function. Another common operation on functions is composition, denoted by the operator $\circ$. If f and g are functions, then $f \circ g$ is also a function, and $f \circ g(x) = f(g(x))$. Closures also make it possible to define $\circ$ as a Lisp function.

Figure 5.3 defines a `compose` function which takes any number of functions and returns their composition. For example

```
(compose #'list #'1+)
```

returns a function equivalent to

```
#'(lambda (x) (list (1+ x)))
```

All the functions given as arguments to `compose` must be functions of one argument, except the last. On the last function there are no restrictions, and whatever arguments it takes, so will the function returned by compose:

```
> (funcall (compose #'1+ #'find-if) #'oddp '(2 3 4))
4
```

```
(defun fif (if then &optional else)
  #'(lambda (x)
      (if (funcall if x)
          (funcall then x)
          (if else (funcall else x)))))

(defun fint (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fint fns)))
        #'(lambda (x)
            (and (funcall fn x) (funcall chain x))))))

(defun fun (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fun fns)))
        #'(lambda (x)
            (or (funcall fn x) (funcall chain x))))))
```

Figure 5.4: More function builders.

Since `not` is a Lisp function, `complement` is a special case of `compose`. It could be defined as:

```
(defun complement (pred)
  (compose #'not pred))
```

We can combine functions in other ways than by composing them. For example, we often see expressions like

```
(mapcar #'(lambda (x)
            (if (slave x)
                (owner x)
                (employer x)))
        people)
```

We could define an operator to build functions like this one automatically. Using `fif` from Figure 5.4, we could get the same effect with:

```
(mapcar (fif #'slave #'owner #'employer)
        people)
```

Figure 5.4 contains several other constructors for commonly occurring types of functions. The second, `fint`, is for cases like this:

```
(find-if #'(lambda (x)
             (and (signed x) (sealed x) (delivered x)))
         docs)
```

The predicate given as the second argument to `find-if` defines the intersection of the three predicates called within it. With `fint`, whose name stands for "function intersection," we can say:

```
(find-if (fint #'signed #'sealed #'delivered) docs)
```

We can define a similar operator to return the union of a set of predicates. The function `fun` is like `fint` but uses `or` instead of `and`.

5.5 Recursion on Cdrs

Recursive functions are so important in Lisp programs that it would be worth having utilities to build them. This section and the next describe functions which build the two most common types. In Common Lisp, these functions are a little awkward to use. Once we get into the subject of macros, we will see how to put a more elegant facade on this machinery. Macros for building recursers are discussed in Sections 15.2 and 15.3.

Repeated patterns in a program are a sign that it could have been written at a higher level of abstraction. What pattern is more commonly seen in Lisp programs than a function like this:

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

or this:

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

Structurally these two functions have a lot in common. They both operate recursively on successive cdrs of a list, evaluating the same expression on each step,

```
(defun lrec (rec &optional base)
  (labels ((self (lst)
             (if (null lst)
                 (if (functionp base)
                     (funcall base)
                     base)
                 (funcall rec (car lst)
                              #'(lambda ()
                                  (self (cdr lst)))))))
    #'self))
```

Figure 5.5: Function to define flat list recursers.

except in the base case, where they return a distinct value. This pattern appears so frequently in Lisp programs that experienced programmers can read and reproduce it without stopping to think. Indeed, the lesson is so quickly learned, that the question of how to package the pattern in a new abstraction does not arise.

However, a pattern it is, all the same. Instead of writing these functions out by hand, we should be able to write a function which will generate them for us. Figure 5.5 contains a function-builder called `lrec` ("list recurser") which should be able to generate most functions that recurse on successive cdrs of a list.

The first argument to `lrec` must be a function of two arguments: the current car of the list, and a function which can be called to continue the recursion. Using `lrec` we could express `our-length` as:

```
(lrec #'(lambda (x f) (1+ (funcall f))) 0)
```

To find the length of the list, we don't need to look at the elements, or stop partway, so the object `x` is always ignored, and the function `f` always called. However, we need to take advantage of both possibilities to express `our-every`, for e.g. `oddp`:[3]

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f))) t)
```

The definition of `lrec` uses `labels` to build a local recursive function called `self`. In the recursive case the function `rec` is passed two arguments, the current car of the list, and a function embodying the recursive call. In functions like `our-every`, where the recursive case is an `and`, if the first argument returns false we want to stop right there. Which means that the argument passed in the recursive

[3]In one widely used Common Lisp, `functionp` erroneously returns true for `t` and `nil`. In that implementation it won't work to give either as the second argument to `lrec`.

```
; copy-list
(lrec #'(lambda (x f) (cons x (funcall f))))

; remove-duplicates
(lrec #'(lambda (x f) (adjoin x (funcall f))))

; find-if, for some function fn
(lrec #'(lambda (x f) (if (fn x) x (funcall f))))

; some, for some function fn
(lrec #'(lambda (x f) (or (fn x) (funcall f))))
```

Figure 5.6: Functions expressed with `lrec`.

case must not be a value but a function, which we can call (if we want) in order to get a value.

Figure 5.6 shows some existing Common Lisp functions defined with `lrec`.[4] Calling `lrec` will not always yield the most efficient implementation of a given function. Indeed, `lrec` and the other recurser generators to be defined in this chapter tend to lead one away from tail-recursive solutions. For this reason they are best suited for use in initial versions of a program, or in parts where speed is not critical.

5.6 Recursion on Subtrees

There is another recursive pattern commonly found in Lisp programs: recursion on subtrees. This pattern is seen in cases where you begin with a possibly nested list, and want to recurse down both its car and its cdr.

The Lisp list is a versatile structure. Lists can represent, among other things, sequences, sets, mappings, arrays, and trees. There are several different ways to interpret a list as a tree. The most common is to regard the list as a binary tree whose left branch is the car and whose right branch is the cdr. (In fact, this is usually the internal representation of lists.) Figure 5.7 shows three examples of lists and the trees they represent. Each internal node in such a tree corresponds to a dot in the dotted-pair representation of the list, so the tree structure may be

[4] In some implementations, you may have to set `*print-circle*` to `t` before these functions can be displayed.

(a . b)   (a b c)   (a b (c d))

Figure 5.7: Lists as trees.

easier to interpret if the lists are considered in that form:

```
(a b c)      =  (a . (b . (c . nil)))
(a b (c d))  =  (a . (b . ((c . (d . nil)) . nil)))
```

Any list can be interpreted as a binary tree. Hence the distinction between pairs of Common Lisp functions like `copy-list` and `copy-tree`. The former copies a list as a sequence—if the list contains sublists, the sublists, being mere elements in the sequence, are not copied:

```
> (setq x     '(a b)
        listx (list x 1))
((A B) 1)
> (eq x (car (copy-list listx)))
T
```

In contrast, `copy-tree` copies a list as a tree—sublists are subtrees, and so must also be copied:

```
> (eq x (car (copy-tree listx)))
NIL
```

We could define a version of `copy-tree` as follows:

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

This definition turns out to be one instance of a common pattern. (Some of the following functions are written a little oddly in order to make the pattern obvious.) Consider for example a utility to count the number of leaves in a tree:

```
(defun count-leaves (tree)
  (if (atom tree)
      1
      (+ (count-leaves (car tree))
         (or (if (cdr tree) (count-leaves (cdr tree)))
             1))))
```

A tree has more leaves than the atoms you can see when it is represented as a list:

```
> (count-leaves '((a b (c d)) (e) f))
10
```

The leaves of a tree are all the atoms you can see when you look at the tree in its *dotted-pair* representation. In dotted-pair notation, `((a b (c d)) (e) f)` would have four `nil`s that aren't visible in the list representation (one for each pair of parentheses) so `count-leaves` returns `10`.

In the last chapter we defined several utilities which operate on trees. For example, `flatten` (page 47) takes a tree and returns a list of all the atoms in it. That is, if you give `flatten` a nested list, you'll get back a list that looks the same except that it's missing all but the outermost pair of parentheses:

```
> (flatten '((a b (c d)) (e) f ()))
(A B C D E F)
```

This function could also be defined (somewhat inefficiently) as follows:

```
(defun flatten (tree)
  (if (atom tree)
      (mklist tree)
      (nconc (flatten (car tree))
             (if (cdr tree) (flatten (cdr tree))))))
```

Finally, consider `rfind-if`, a recursive version of `find-if` which works on trees as well as flat lists:

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (if (cdr tree) (rfind-if fn (cdr tree))))))
```

To generalize `find-if` for trees, we have to decide whether we want to search for just leaves, or for whole subtrees. Our `rfind-if` takes the former approach, ○ so the caller can assume that the function given as the first argument will only be called on atoms:

```
> (rfind-if (fint #'numberp #'oddp) '(2 (3 4) 5))
3
```

How similar in form are these four functions, `copy-tree`, `count-leaves`, `flatten`, and `rfind-if`. Indeed, they're all instances of an archetypal function for recursion on subtrees. As with recursion on cdrs, we need not leave this archetype to float vaguely in the background—we can write a function to generate instances of it.

To get at the archetype itself, let's look at these functions and see what's not pattern. Essentially `our-copy-tree` is two facts:

1. In the base case it returns its argument.

2. In the recursive case, it applies `cons` to the recursions down the left (car) and right (cdr) subtrees.

We should thus be able to express it as a call to a builder with two arguments:

```
(ttrav #'cons #'identity)
```

A definition of `ttrav` ("tree traverser") is shown in Figure 5.8. Instead of passing one value in the recursive case, we pass two, one for the left subtree and one for the right. If the `base` argument is a function it will be called on the current leaf. In flat list recursion, the base case is always `nil`, but in tree recursion the base case could be an interesting value, and we might want to use it.

With `ttrav` we could express all the preceding functions except `rfind-if`. (They are shown in Figure 5.9.) To define `rfind-if` we need a more general tree recursion builder which gives us control over when, and if, the recursive calls are made. As the first argument to `ttrav` we gave a function which took the *results* of the recursive calls. For the general case, we want to use instead a function which takes two closures representing the calls themselves. Then we can write recursers which only traverse as much of the tree as they want to.

```
(defun ttrav (rec &optional (base #'identity))
  (labels ((self (tree)
             (if (atom tree)
                 (if (functionp base)
                     (funcall base tree)
                     base)
                 (funcall rec (self (car tree))
                              (if (cdr tree)
                                  (self (cdr tree)))))))
    #'self))
```

Figure 5.8: Function for recursion on trees.

```
; our-copy-tree
(ttrav #'cons)

; count-leaves
(ttrav #'(lambda (l r) (+ l (or r 1)))  1)

; flatten
(ttrav #'nconc #'mklist)
```

Figure 5.9: Functions expressed with `ttrav`.

Functions built by `ttrav` always traverse a whole tree. That's fine for functions like `count-leaves` or `flatten`, which have to traverse the whole tree anyway. But we want `rfind-if` to stop searching as soon as it finds what it's looking for. It must be built by the more general `trec`, shown in Figure 5.10. The second arg to `trec` should be a function of three arguments: the current object and the two recursers. The latter two will be closures representing the recursions down the left and right subtrees. With `trec` we would define `flatten` as:

```
(trec #'(lambda (o l r) (nconc (funcall l) (funcall r)))
      #'mklist)
```

Now we can also express `rfind-if` for e.g. `oddp` as:

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

```
(defun trec (rec &optional (base #'identity))
  (labels
    ((self (tree)
       (if (atom tree)
           (if (functionp base)
               (funcall base tree)
               base)
           (funcall rec tree
                        #'(lambda ()
                            (self (car tree)))
                        #'(lambda ()
                            (if (cdr tree)
                                (self (cdr tree))))))))
    #'self))
```

Figure 5.10: Function for recursion on trees.

5.7 When to Build Functions

Expressing functions by calls to constructors instead of sharp-quoted lambda-expressions could, unfortunately, entail unnecessary work at runtime. A sharp-quoted lambda-expression is a constant, but a call to a constructor function will be evaluated at runtime. If we really have to make this call at runtime, it might not be worth using constructor functions. However, at least some of the time we can call the constructor beforehand. By using `#.`, the sharp-dot read macro, we can have the new functions built at read-time. So long as `compose` and its arguments are defined when this expression is read, we could say, for example,

```
(find-if #.(compose #'oddp #'truncate) lst)
```

Then the call to `compose` would be evaluated by the reader, and the resulting function inserted as a constant into our code. Since both `oddp` and `truncate` are built-in, it would safe to assume that we can evaluate the `compose` at read-time, so long as `compose` itself were already loaded.

In general, composing and combining functions is more easily and efficiently done with macros. This is particularly true in Common Lisp, with its separate name-space for functions. After introducing macros, we will in Chapter 15 cover much of the ground we covered here, but in a more luxurious vehicle.

6

Functions as Representation

Generally, data structures are used to *represent.* An array could represent a geometric transformation; a tree could represent a hierarchy of command; a graph could represent a rail network. In Lisp we can sometimes use closures as a representation. Within a closure, variable bindings can store information, and can also play the role that pointers play in constructing complex data structures. By making a group of closures which share bindings, or can refer to one another, we can create hybrid objects which combine the advantages of data structures and programs.

Beneath the surface, shared bindings *are* pointers. Closures just bring us the convenience of dealing with them at a higher level of abstraction. By using closures to represent something we would otherwise represent with static data structures, we can often expect substantial improvements in elegance and efficiency.

6.1 Networks

Closures have three useful properties: they are active, they have local state, and we can make multiple instances of them. Where could we use multiple copies of active objects with local state? In applications involving networks, among others. In many cases we can represent nodes in a network as closures. As well as having its own local state, a closure can refer to another closure. Thus a closure representing a node in a network can know of several other nodes (closures) to which it must send its output. This means that we may be able to translate some networks straight into code.

```
> (run-node 'people)
Is the person a man?
>> yes
Is he living?
>> no
Was he American?
>> yes
Is he on a coin?
>> yes
Is the coin a penny?
>> yes
LINCOLN
```

Figure 6.1: Session of twenty questions.

In this section and the next we will look at two ways to traverse a network. First we will follow the traditional approach, with nodes defined as structures, and separate code to traverse the network. Then in the next section we'll show how to build the same program from a single abstraction.

As an example, we will use about the simplest application possible: one of those programs that play twenty questions. Our network will be a binary tree. Each non-leaf node will contain a yes/no question, and depending on the answer to the question, the traversal will continue down the left or right subtree. Leaf nodes will contain return values. When the traversal reaches a leaf node, its value will be returned as the value of the traversal. A session with this program might look as in Figure 6.1.

The traditional way to begin would be to define some sort of data structure to represent nodes. A node is going to have to know several things: whether it is a leaf; if so, which value to return, and if not, which question to ask; and where to go depending on the answer. A sufficient data structure is defined in Figure 6.2. It is designed for minimal size. The `contents` field will contain either a question or a return value. If the node is not a leaf, the `yes` and `no` fields will tell where to go depending on the answer to the question; if the node is a leaf, we will know it because these fields are empty. The global `*nodes*` will be a hash-table in which nodes are indexed by name. Finally, `defnode` makes a new node (of either type) and stores it in `*nodes*`. Using these materials we could define the first node of our tree:

```
(defnode 'people "Is the person a man?"
         'male 'female)
```

```
(defstruct node  contents yes no)

(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (make-node :contents conts
                   :yes      yes
                   :no       no)))
```

Figure 6.2: Representation and definition of nodes.

```
(defnode 'people "Is the person a man?" 'male 'female)

(defnode 'male "Is he living?" 'liveman 'deadman)

(defnode 'deadman "Was he American?" 'us 'them)

(defnode 'us "Is he on a coin?" 'coin 'cidence)

(defnode 'coin "Is the coin a penny?" 'penny 'coins)

(defnode 'penny 'lincoln)
```

Figure 6.3: Sample network.

Figure 6.3 shows as much of the network as we need to produce the transcript in Figure 6.1.

Now all we need to do is write a function to traverse this network, printing out the questions and following the indicated path. This function, `run-node`, is shown in Figure 6.4. Given a name, we look up the corresponding node. If it is not a leaf, the `contents` are asked as a question, and depending on the answer, we continue traversing at one of two possible destinations. If the node is a leaf, `run-node` just returns its `contents`. With the network defined in Figure 6.3, this function produces the output shown in Figure 6.1.

```
(defun run-node (name)
  (let ((n (gethash name *nodes*)))
    (cond ((node-yes n)
           (format t "~A~%>> " (node-contents n))
           (case (read)
             (yes (run-node (node-yes n)))
             (t   (run-node (node-no n)))))
          (t (node-contents n)))))
```

Figure 6.4: Function for traversing networks.

```
(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (if yes
            #'(lambda ()
                (format t "~A~%>> " conts)
                (case (read)
                  (yes (funcall (gethash yes *nodes*)))
                  (t   (funcall (gethash no  *nodes*)))))
            #'(lambda () conts))))
```

Figure 6.5: A network compiled into closures.

6.2 Compiling Networks

In the preceding section we wrote a network program as it might have been written in any language. Indeed, the program is so simple that it seems odd to think that we could write it any other way. But we can—in fact, we can write it much more simply.

The code in Figure 6.5 illustrates this point. It's all we really need to run our network. Instead of having nodes as data structures and a separate function to traverse them, we represent the nodes as closures. The data formerly contained in the structures gets stored in variable bindings within the closures. Now there is no need for `run-node`; it is implicit in the nodes themselves. To start the traversal,

```
(defvar *nodes* nil)

(defun defnode (&rest args)
  (push args *nodes*)
  args)

(defun compile-net (root)
  (let ((node (assoc root *nodes*)))
    (if (null node)
        nil
        (let ((conts (second node))
              (yes (third node))
              (no (fourth node)))
          (if yes
              (let ((yes-fn (compile-net yes))
                    (no-fn  (compile-net no)))
                #'(lambda ()
                    (format t "~A~%>> " conts)
                    (funcall (if (eq (read) 'yes)
                                 yes-fn
                                 no-fn))))
              #'(lambda () conts))))))
```

Figure 6.6: Compilation with static references.

we just funcall the node at which we want to begin:

```
(funcall (gethash 'people *nodes*))
Is the person a man?
>>
```

From then on, the transcript will be just as it was with the previous implementation.

By representing the nodes as closures, we are able to transform our twenty-questions network entirely into code. As it is, the code will have to look up the node functions by name at runtime. However, if we know that the network is not going to be redefined on the fly, we can add a further enhancement: we can have node functions call their destinations directly, without having to go through a hash-table.

Figure 6.6 contains a new version of the program. Now `*nodes*` is a disposable list instead of a hash-table. All the nodes are defined with `defnode` as before, but no closures are generated at this point. After all the nodes have been defined, we call compile-net to compile a whole network at once. This function recursively works its way right down to the leaves of the tree, and on the way back up, returns at each step the node/function for each of the two subtrees.[1] So now each node will have a direct handle on its two destinations, instead of having only their names. When the original call to compile-net returns, it will yield a function representing the portion of the network we asked to have compiled.

```
> (setq n (compile-net 'people))
#<Compiled-Function BF3C06>
> (funcall n)
Is the person a man?
>>
```

Notice that compile-net compiles in both senses. It compiles in the general sense, by translating the abstract representation of the network into code. Moreover, if compile-net itself is compiled, it will return compiled functions. (See page 25.)

After compiling the network, we will no longer need the list made by defnode. It can be cut loose (e.g. by setting *nodes* to nil) and reclaimed by the garbage collector.

6.3 Looking Forward

Many programs involving networks can be implemented by compiling the nodes into closures. Closures are data objects, and they can be used to represent things just as structures can. Doing so requires some unconventional thinking, but the rewards are faster and more elegant programs.

Macros help substantially when we use closures as a representation. "To represent with closures" is another way of saying "to compile," and since macros do their work at compile-time, they are a natural vehicle for this technique. After macros have been introduced, Chapters 23 and 24 will present much larger programs based on the strategy used here.

[1] This version assumes that the network is a tree, which it must be in this application.

7

Macros

Lisp's macro facility allows you to define operators that are implemented by transformation. The definition of a macro is essentially a function that generates Lisp code—a program that writes programs. From these small beginnings arise great possibilities, and also unexpected hazards. Chapters 7–10 form a tutorial on macros. This chapter explains how macros work, gives techniques for writing and testing them, and looks at the issue of macro style.

7.1 How Macros Work

Since macros can be called and return values, they tend to be associated with functions. Macro definitions sometimes resemble function definitions, and speaking informally, people call `do`, which is actually a macro, a "built-in function." But pushing the analogy too far can be a source of confusion. Macros work differently from normal functions, and knowing how and why macros are different is the key to using them correctly. A function produces results, but a macro produces *expressions*—which, when evaluated, produce results.

The best way to begin is to move straight into an example. Suppose we want to write a macro `nil!`, which sets its argument to `nil`. We want `(nil! x)` to have the same effect as `(setq x nil)`. We do it by defining `nil!` as a macro which turns instances of the first form into instances of the second.

```
> (defmacro nil! (var)
    (list 'setq var nil))
NIL!
```

Paraphrased in English, this definition tells Lisp: "Whenever you see an expression of the form `(nil!` *var*`)`, turn it into one of the form `(setq` *var* `nil)` before evaluating it."

The expression generated by the macro will be evaluated in place of the original macro call. A macro call is a list whose first element is the name of a macro. What happens when we type the macro call `(nil! x)` into the toplevel? Lisp notices that `nil!` is the name of a macro, and

1. builds the expression specified by the definition above, then
2. evaluates that expression in place of the original macro call.

The step of building the new expression is called *macroexpansion.* Lisp looks up the definition of `nil!`, which shows how to construct a replacement for the macro call. The definition of `nil!` is applied like a function to the *expressions* given as arguments in the macro call. It returns a list of three elements: `setq`, the expression given as the argument to the macro, and `nil`. In this case, the argument to `nil!` is `x`, and the macroexpansion is `(setq x nil)`.

After macroexpansion comes a second step, *evaluation.* Lisp evaluates the macroexpansion `(setq x nil)` as if you had typed that in the first place. Evaluation does not always come immediately after expansion, as it does at the toplevel. A macro call occurring in the definition of a function will be expanded when the function is compiled, but the expansion—or the object code which results from it—won't be evaluated until the function is called.

Many of the difficulties you might encounter with macros can be avoided by maintaining a sharp distinction between macroexpansion and evaluation. When writing macros, know which computations are performed during macroexpansion, and which during evaluation, for the two steps generally operate on objects of two different sorts. The macroexpansion step deals with expressions, and the evaluation step deals with their values.

Sometimes macroexpansion can be more complicated than it was in the case of `nil!`. The expansion of `nil!` was a call to a built-in special form, but sometimes the expansion of a macro will be yet another macro call, like a Russian doll which contains another doll inside it. In such cases, macroexpansion simply continues until it arrives at an expression which is no longer a macro call. The process can take arbitrarily many steps, so long as it terminates eventually.

Many languages offer some form of macro, but Lisp macros are singularly powerful. When a file of Lisp is compiled, a parser reads the source code and sends its output to the compiler. Here's the stroke of genius: the output of the parser consists of *lists of Lisp objects.* With macros, we can manipulate the program while it's in this intermediate form between parser and compiler. If necessary, these manipulations can be very extensive. A macro generating its expansion has at its disposition the full power of Lisp. Indeed, a macro is really a Lisp function—one which happens to return expressions. The definition of `nil!` contains a single call to `list`, but another macro might invoke a whole subprogram to generate its expansion.

Being able to change what the compiler sees is almost like being able to rewrite it. We can add any construct to the language that we can define by transformation into existing constructs.

7.2 Backquote

Backquote is a special version of quote which can be used to create templates for Lisp expressions. One of the most common uses of backquote is in macro definitions.

The backquote character, `` ` ``, is so named because it resembles a regular quote, `'`, reversed. When backquote alone is affixed to an expression, it behaves just like quote:

`` `(a b c) `` is equal to `'(a b c)`.

Backquote becomes useful only when it appears in combination with comma, `,`, and comma-at, `,@`. If backquote makes a template, comma makes a slot within a template. A backquoted list is equivalent to a call to `list` with the elements quoted. That is,

`` `(a b c) `` is equal to `(list 'a 'b 'c)`.

Within the scope of a backquote, a comma tells Lisp: "turn off the quoting." When a comma appears before one of the elements of the list, it has the effect of cancelling out the quote that would have been put there. So

`` `(a ,b c ,d) `` is equal to `(list 'a b 'c d)`.

Instead of the symbol `b`, its value is inserted into the resulting list. Commas work no matter how deeply they appear within a nested list,

```
> (setq a 1 b 2 c 3)
3
> `(a ,b c)
(A 2 C)
> `(a (,b c))
(A (2 C))
```

and they may even appear within quotes, or within quoted sublists:

```
> `(a b ,c (',(+ a b c)) (+ a b) 'c '((,a ,b)))
(A B 3 ('6) (+ A B) 'C '((1 2)))
```

One comma counteracts the effect of one backquote, so commas must match backquotes. Say that a comma is *surrounded* by a particular operator if the operator is prepended to the comma, or prepended to an expression which contains it. In `` `(,a ,(b `,c)))``, for example, the last comma is surrounded by one comma and two backquotes. The general rule is: a comma surrounded by n commas must be surrounded by at least $n+1$ backquotes. An obvious corollary is that commas may not appear outside of a backquoted expression. Backquotes and commas can be nested, so long as they obey the rule above. Any of the following expressions would generate an error if typed into the toplevel:

```
,x    `(a ,,b c)    `(a ,(b ,c) d)    `(,,`a)
```

Nested backquotes are only likely to be needed in macro-defining macros. Both topics are discussed in Chapter 16.

Backquote is usually used for making lists.[1] Any list generated by backquote can also be generated by using `list` and regular quotes. The advantage of backquote is just that it makes expressions easier to read, because a backquoted expression resembles the expression it will produce. In the previous section we defined `nil!` as:

```
(defmacro nil! (var)
  (list 'setq var nil))
```

With backquote the same macro can be defined as:

```
(defmacro nil! (var)
  `(setq ,var nil))
```

which in this case is not all that different. The longer the macro definition, however, the more important it is to use backquote. Figure 7.1 contains two possible definitions of `nif`, a macro which does a three-way numeric `if`.[2]

The first argument should evaluate to a number. Then the second, third, or fourth argument is evaluated, depending on whether the first was positive, zero, or negative:

```
> (mapcar #'(lambda (x)
              (nif x 'p 'z 'n))
          '(0 2.5 -8))
(Z P N)
```

With backquote:

```
(defmacro nif (expr pos zero neg)
  `(case (truncate (signum ,expr))
     (1 ,pos)
     (0 ,zero)
     (-1 ,neg)))
```

Without backquote:

```
(defmacro nif (expr pos zero neg)
  (list 'case
        (list 'truncate (list 'signum expr))
        (list 1 pos)
        (list 0 zero)
        (list -1 neg)))
```

Figure 7.1: A macro defined with and without backquote.

The two definitions in Figure 7.1 define the same macro, but the first uses backquote, while the second builds its expansion by explicit calls to `list`. From the first definition it's easy to see that `(nif x 'p 'z 'n)`, for example, expands into

```
(case (truncate (signum x))
  (1 'p)
  (0 'z)
  (-1 'n))
```

because the body of the macro definition looks just like the expansion it generates. To understand the second version, without backquote, you have to trace in your head the building of the expansion.

Comma-at, `,@`, is a variant of comma. It behaves like comma, with one difference: instead of merely inserting the value of the expression to which it is affixed, as comma does, comma-at *splices* it. Splicing can be thought of as inserting while removing the outermost level of parentheses:

```
> (setq b '(1 2 3))
(1 2 3)
```

[1]Backquote can also be used to create vectors, but this is rarely done in macro definitions.

[2]This macro is defined a little oddly to avoid using gensyms. A better definition is given on page 150.

```
> `(a ,b c)
(A (1 2 3) C)
> `(a ,@b c)
(A 1 2 3 C)
```

The comma causes the list `(1 2 3)` to be inserted in place of `b`, while the comma-at causes the elements of the list to be inserted there. There are some additional restrictions on the use of comma-at:

1. In order for its argument to be spliced, comma-at must occur within a sequence. It's an error to say something like `` `,@b`` because there is nowhere to splice the value of `b`.

2. The object to be spliced must be a list, unless it occurs last. The expression `` `(a ,@1)`` will evaluate to `(a . 1)`, but attempting to splice an atom into the middle of a list, as in `` `(a ,@1 b)``, will cause an error.

Comma-at tends to be used in macros which take an indeterminate number of arguments and pass them on to functions or macros which also take an indeterminate number of arguments. This situation commonly arises when implementing implicit blocks. Common Lisp has several operators for grouping code into blocks, including `block`, `tagbody`, and `progn`. These operators rarely appear directly in source code; they are more often *implicit*—that is, hidden by macros.

An implicit block occurs in any built-in macro which can have a *body* of expressions. Both `let` and `cond` provide implicit `progn`, for example. The simplest built-in macro to do so is probably `when`:

```
(when (eligible obj)
  (do-this)
  (do-that)
  obj)
```

If `(eligible obj)` returns true, the remaining expressions will be evaluated, and the `when` expression as a whole will return the value of the last. As an example of the use of comma-at, here is one possible definition for `when`:

```
(defmacro our-when (test &body body)
  `(if ,test
       (progn
         ,@body)))
```

This definition uses an `&body` parameter (identical to `&rest` except for its effect on pretty-printing) to take in an arbitrary number of arguments, and a comma-at to splice them into a `progn` expression. In the macroexpansion of the call above, the three expressions in the body will appear within a single `progn`:

```
(if (eligible obj)
    (progn (do-this)
           (do-that)
           obj))
```

Most macros for iteration splice their arguments in a similar way.

The effect of comma-at can be achieved without using backquote. The expression `‘(a ,@b c)` is equal to `(cons ’a (append b (list ’c)))`, for example. Comma-at exists only to make such expression-generating expressions more readable.

Macro definitions (usually) generate lists. Although macro expansions could be built with the function `list`, backquote list-templates make the task much easier. A macro defined with `defmacro` and backquote will superficially resemble a function defined with `defun`. So long as you are not misled by the similarity, backquote makes macro definitions both easier to write and easier to read.

Backquote is so often used in macro definitions that people sometimes think of backquote as part of `defmacro`. The last thing to remember about backquote is that it has a life of its own, separate from its role in macros. You can use backquote anywhere sequences need to be built:

```
(defun greet (name)
  ‘(hello ,name))
```

7.3 Defining Simple Macros

In programming, the best way to learn is often to begin experimenting as soon as possible. A full theoretical understanding can come later. Accordingly, this section presents a way to start writing macros immediately. It works only for a narrow range of cases, but where applicable it can be applied quite mechanically. (If you’ve written macros before, you may want to skip this section.)

As an example, we consider how to write a variant of the the built-in Common Lisp function `member`. By default `member` uses `eql` to test for equality. If you want to test for membership using `eq`, you have to say so explicitly:

```
(member x choices :test #’eq)
```

If we did this a lot, we might want to write a variant of `member` which always used `eq`. Some earlier dialects of Lisp had such a function, called `memq`:

```
(memq x choices)
```

Ordinarily one would define `memq` as an inline function, but for the sake of example we will reincarnate it as a macro.

```
call:            (memq x choices)

expansion:  (member x choices :test #'eq)
```

Figure 7.2: Diagram used in writing memq.

The method: Begin with a typical call to the macro you want to define. Write it down on a piece of paper, and below it write down the expression into which it ought to expand. Figure 7.2 shows two such expressions. From the macro call, construct the parameter list for your macro, making up some parameter name for each of the arguments. In this case there are two arguments, so we'll have two parameters, and call them `obj` and `lst`:

```
(defmacro memq (obj lst)
```

Now go back to the two expressions you wrote down. For each argument in the macro call, draw a line connecting it with the place it appears in the expansion below. In Figure 7.2 there are two parallel lines. To write the body of the macro, turn your attention to the expansion. Start the body with a backquote. Now, begin reading the expansion expression by expression. Wherever you find a parenthesis that isn't part of an argument in the macro call, put one in the macro definition. So following the backquote will be a left parenthesis. For each expression in the expansion

1. If there is no line connecting it with the macro call, then write down the expression itself.

2. If there is a connection to one of the arguments in the macro call, write down the symbol which occurs in the corresponding position in the macro parameter list, preceded by a comma.

There is no connection to the first element, `member`, so we use `member` itself:

```
(defmacro memq (obj lst)
  `(member
```

However, `x` has a line leading to the first argument in the source expression, so we use in the macro body the first parameter, with a comma:

```
(defmacro memq (obj lst)
  `(member ,obj
```

Continuing in this way, the completed macro definition is:

```
(while hungry
  (stare-intently)
  (meow)
  (rub-against-legs))

(do ()
    ((not hungry))
  (stare-intently)
  (meow)
  (rub-against-legs))
```

Figure 7.3: Diagram used in writing `while`.

```
(defmacro memq (obj lst)
  `(member ,obj ,lst :test #'eq))
```

So far, we can only write macros which take a fixed number of arguments. Now suppose we want to write a macro `while`, which will take a test expression and some body of code, and loop through the code as long as the test expression returns true. Figure 7.3 contains an example of a `while` loop describing the behavior of a cat.

To write such a macro, we have to modify our technique slightly. As before, begin by writing down a sample macro call. From that, build the parameter list of the macro, but where you want to take an indefinite number of arguments, conclude with an `&rest` or `&body` parameter:

```
(defmacro while (test &body body)
```

Now write the desired expansion below the macro call, and as before draw lines connecting the arguments in the macro call to their position in the expansion. However, when you have a sequence of arguments which are going to be sucked into a single `&rest` or `&body` parameter, treat them as a group, drawing a single line for the whole sequence. Figure 7.3 shows the resulting diagram.

To write the body of the macro definition, proceed as before along the expansion. As well as the two previous rules, we need one more:

3. If there is a connection from a series of expressions in the expansion to a series of the arguments in the macro call, write down the corresponding `&rest` or `&body` parameter, preceded by a comma-at.

So the resulting macro definition will be:

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

To build a macro which can have a *body* of expressions, some parameter has to act as a funnel. Here multiple arguments in the macro call are joined together into `body`, and then broken up again when `body` is spliced into the expansion.

The approach described in this section enables us to write the simplest macros—those which merely shuffle their parameters. Macros can do a lot more than that. Section 7.7 will present examples where expansions can't be represented as simple backquoted lists, and to generate them, macros become programs in their own right.

7.4 Testing Macroexpansion

Having written a macro, how do we test it? A macro like `memq` is simple enough that one can tell just by looking at it what it will do. When writing more complicated macros, we have to be able to check that they are being expanded correctly.

Figure 7.4 shows a macro definition and two ways of looking at its expansion. The built-in function `macroexpand` takes an expression and returns its macroexpansion. Sending a macro call to `macroexpand` shows how the macro call will finally be expanded before being evaluated, but a complete expansion is not always what you want in order to test a macro. When the macro in question relies on other macros, they too will be expanded, so a complete macroexpansion can sometimes be difficult to read.

From the first expression shown in Figure 7.4, it's hard to tell whether or not `while` is expanding as intended, because the built-in `do` macro gets expanded, as well as the `prog` macro into which *it* expands. What we need is a way of seeing the result after only one step of expansion. This is the purpose of the built-in function `macroexpand-1`, shown in the second example; `macroexpand-1` stops after just one step, even if the expansion is still a macro call.

When we want to look at the expansion of a macro call, it will be a nuisance always to have to type

```
(pprint (macroexpand-1 '(or x y)))
```

Figure 7.5 defines a new macro which allows us to say instead:

```
(mac (or x y))
```

Typically you debug functions by calling them, and macros by expanding them. But since a macro call involves two layers of computation, there are two

```
> (defmacro while (test &body body)
    `(do ()
         ((not ,test))
       ,@body))
WHILE

> (pprint (macroexpand '(while (able) (laugh))))

(BLOCK NIL
  (LET NIL
    (TAGBODY
      #:G61
      (IF (NOT (ABLE)) (RETURN NIL))
      (LAUGH)
      (GO #:G61))))
T
> (pprint (macroexpand-1 '(while (able) (laugh))))

(DO NIL
    ((NOT (ABLE)))
  (LAUGH))
T
```

Figure 7.4: A macro and two depths of expansion.

```
(defmacro mac (expr)
  `(pprint (macroexpand-1 ',expr)))
```

Figure 7.5: A macro for testing macroexpansion.

points where things can go wrong. If a macro is misbehaving, most of the time you will be able to tell what's wrong just by looking at the expansion. Sometimes, though, the expansion will look fine and you'll want to evaluate it to see where the problems arise. If the expansion contains free variables, you may want to set some variables first. In some systems, you will be able to copy the expansion and paste it into the toplevel, or select it and choose `eval` from a menu. In the worst case you can set a variable to the list returned by `macroexpand-1`, then call `eval` on it:

```
> (setq exp (macroexpand-1 '(memq 'a '(a b c))))
(MEMBER (QUOTE A) (QUOTE (A B C)) :TEST (FUNCTION EQ))
> (eval exp)
(A B C)
```

Finally, macroexpansion is more than an aid in debugging, it's also a way of learning how to write macros. Common Lisp has over a hundred macros built-in, some of them quite complex. By looking at the expansions of these macros you will often be able to see how they were written.

7.5 Destructuring in Parameter Lists

Destructuring is a generalization of the sort of assignment[3] done by function calls. If you define a function of several arguments

```
(defun foo (x y z)
  (+ x y z))
```

then when the function is called

```
(foo 1 2 3)
```

the parameters of the function are assigned arguments in the call according to their position: `x` to `1`, `y` to `2`, and `z` to `3`. *Destructuring* describes the situation where this sort of positional assignment is done for arbitrary list structures, as well as flat lists like `(x y z)`.

The Common Lisp `destructuring-bind` macro (new in CLTL2) takes a pattern, an argument evaluating to a list, and a body of expressions, and evaluates the expressions with the parameters in the pattern bound to the corresponding elements of the list:

```
> (destructuring-bind (x (y) . z) '(a (b) c d)
    (list x y z))
(A B (C D))
```

This new operator and others like it form the subject of Chapter 18.

Destructuring is also possible in macro parameter lists. The Common Lisp `defmacro` allows parameter lists to be arbitrary list structures. When a macro call is expanded, components of the call will be assigned to the parameters as if by `destructuring-bind`. The built-in `dolist` macro takes advantage of such parameter list destructuring. In a call like:

[3] Destructuring is usually seen in operators which create bindings, rather than do assignments. However, conceptually destructuring is a way of assigning values, and would work just as well for existing variables as for new ones. That is, there is nothing to stop you from writing a destructuring `setq`.

```
(dolist (x '(a b c))
  (print x))
```

the expansion function must pluck `x` and `'(a b c)` from within the list given as the first argument. That can be done implicitly by giving `dolist` the appropriate parameter list:[4]

```
(defmacro our-dolist ((var list &optional result) &body body)
  `(progn
     (mapc #'(lambda (,var) ,@body)
           ,list)
     (let ((,var nil))
       ,result)))
```

In Common Lisp, macros like `dolist` usually enclose within a list the arguments not part of the body. Because it takes an optional `result` argument, `dolist` must enclose its first arguments in a distinct list anyway. But even if the extra list structure were not necessary, it would make calls to `dolist` easier to read. Suppose we want to define a macro `when-bind`, like `when` except that it binds some variable to the value returned by the test expression. This macro may be best implemented with a nested parameter list:

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))
```

and called as follows:

```
(when-bind (input (get-user-input))
  (process input))
```

instead of:

```
(let ((input (get-user-input)))
  (when input
    (process input)))
```

Used sparingly, parameter list destructuring can result in clearer code. At a minimum, it can be used in macros like `when-bind` and `dolist`, which take two or more arguments followed by a body of expressions.

[4]This version is written in this strange way to avoid using gensyms, which are not introduced till later.

```
(defmacro our-expander (name) `(get ,name 'expander))

(defmacro our-defmacro (name parms &body body)
  (let ((g (gensym)))
    `(progn
       (setf (our-expander ',name)
             #'(lambda (,g)
                 (block ,name
                   (destructuring-bind ,parms (cdr ,g)
                     ,@body))))
       ',name)))

(defun our-macroexpand-1 (expr)
  (if (and (consp expr) (our-expander (car expr)))
      (funcall (our-expander (car expr)) expr)
      expr))
```

Figure 7.6: A sketch of `defmacro`.

7.6 A Model of Macros

A formal description of what macros do would be long and confusing. Experienced programmers do not carry such a description in their heads anyway. It's more convenient to remember what `defmacro` does by imagining how it would be defined.

There is a long tradition of such explanations in Lisp. The *Lisp 1.5 Programmer's Manual*, first published in 1962, gives for reference a definition of `eval` written in Lisp. Since `defmacro` is itself a macro, we can give it the same treatment, as in Figure 7.6. This definition uses several techniques which haven't been covered yet, so some readers may want to refer to it later. ○

The definition in Figure 7.6 gives a fairly accurate impression of what macros do, but like any sketch it is incomplete. It wouldn't handle the `&whole` keyword properly. And what `defmacro` really stores as the `macro-function` of its first argument is a function of *two* arguments: the macro call, and the lexical environment in which it occurs. However, these features are used only by the most esoteric macros. If you worked on the assumption that macros were implemented as in Figure 7.6, you would hardly ever go wrong. Every macro defined in this book would work, for example.

The definition in Figure 7.6 yields an expansion function which is a sharp-quoted lambda-expression. That should make it a closure: any free symbols in the macro definition should refer to variables in the environment where the `defmacro` occurred. So it should be possible to say this:

```
(let ((op 'setq))
  (defmacro our-setq (var val)
    (list op var val)))
```

As of CLTL2, it is. But in CLTL1, macro expanders were defined in the null lexical environment,[5] so in some old implementations this definition of `our-setq` will not work.

7.7 Macros as Programs

A macro definition need not be just a backquoted list. A macro is a function which transforms one sort of expression into another. This function can call `list` to generate its result, but can just as well invoke a whole subprogram consisting of hundreds of lines of code.

Section 7.3 gave an easy way of writing macros. Using this technique we can write macros whose expansions contain the same subexpressions as appear in the macro call. Unfortunately, only the simplest macros meet this condition. As a more complicated example, consider the built-in macro `do`. It isn't possible to write `do` as a macro which simply shuffles its parameters. The expansion has to build complex expressions which never appear in the macro call.

The more general approach to writing macros is to think about the sort of expression you want to be able to use, what you want it to expand into, and then write the *program* that will transform the first form into the second. Try expanding an example by hand, then look at what happens when one form is transformed into another. By working from examples you can get an idea of what will be required of your proposed macro.

Figure 7.7 shows an instance of `do`, and the expression into which it should expand. Doing expansions by hand is a good way to clarify your ideas about how a macro should work. For example, it may not be obvious until one tries writing the expansion that the local variables will have to be updated using `psetq`.

The built-in macro `psetq` (named for "parallel `setq`") behaves like `setq`, except that all its (even-numbered) arguments will be evaluated before any of the assignments are made. If an ordinary `setq` has more than two arguments, then the new value of the first argument is visible during the evaluation of the fourth:

[5] For an example of macro where this distinction matters, see the note on page 393.

```
(do ((w 3)
     (x 1 (1+ x))
     (y 2 (1+ y))
     (z))
    ((> x 10) (princ z) y)
  (princ x)
  (princ y))
```

should expand into something like

```
(prog ((w 3) (x 1) (y 2) (z nil))
   foo
    (if (> x 10)
        (return (progn (princ z) y)))
    (princ x)
    (princ y)
    (psetq x (1+ x) y (1+ y))
   (go foo))
```

Figure 7.7: Desired expansion of `do`.

```
> (let ((a 1))
    (setq a 2 b a)
    (list a b))
(2 2)
```

Here, because `a` is set first, `b` gets its new value, `2`. A `psetq` is supposed to behave as if its arguments were assigned in parallel:

```
> (let ((a 1))
    (psetq a 2 b a)
    (list a b))
(2 1)
```

So here `b` gets the old value of `a`. The `psetq` macro is provided especially to support macros like `do`, which need to evaluate some of their arguments in parallel. (Had we used `setq`, we would have been defining `do*` instead.)

On looking at the expansion, it is also clear that we can't really use `foo` as the loop label. What if `foo` is also used as a loop label within the body of the `do`? Chapter 9 will deal with this problem in detail; for now, suffice it to say that instead of using `foo`, the macroexpansion must use a special anonymous symbol returned by the function `gensym`.

```
(defmacro our-do (bindforms (test &rest result) &body body)
  (let ((label (gensym)))
    `(prog ,(make-initforms bindforms)
       ,label
       (if ,test
           (return (progn ,@result)))
       ,@body
       (psetq ,@(make-stepforms bindforms))
       (go ,label))))

(defun make-initforms (bindforms)
  (mapcar #'(lambda (b)
              (if (consp b)
                  (list (car b) (cadr b))
                  (list b nil)))
          bindforms))

(defun make-stepforms (bindforms)
  (mapcan #'(lambda (b)
              (if (and (consp b) (third b))
                  (list (car b) (third b))
                  nil))
          bindforms))
```

Figure 7.8: Implementing `do`.

In order to write `do`, we consider what it would take to transform the first expression in Figure 7.7 into the second. To perform such a transformation, we need to do more than get the macro parameters into the right positions in some backquoted list. The initial `prog` has to be followed by a list of symbols and their initial bindings, which must be extracted from the second argument passed to the `do`. The function `make-initforms` in Figure 7.8 will return such a list. We also have to build a list of arguments for the `psetq`, but this case is more complicated because not all the symbols should be updated. In Figure 7.8, `make-stepforms` returns arguments for the `psetq`. With these two functions, the rest of the definition becomes fairly straightforward.

The code in Figure 7.8 isn't exactly the way `do` would be written in a real implementation. To emphasize the computation done during expansion, `make-initforms` and `make-stepforms` have been broken out as separate functions. In the future, such code will usually be left within the `defmacro` expression.

With the definition of this macro, we begin to see what macros can do. A macro has full access to Lisp to build an expansion. The code used to generate the expansion may be a program in its own right.

7.8 Macro Style

Good style means something different for macros. Style matters when code is either read by people or evaluated by Lisp. With macros, both of these activities take place under slightly unusual circumstances.

There are two different kinds of code associated with a macro definition: *expander code*, the code used by the macro to generate its expansion, and *expansion code*, which appears in the expansion itself. The principles of style are different for each. For programs in general, to have good style is to be clear and efficient. These principles are bent in opposite directions by the two types of macro code: expander code can favor clarity over efficiency, and expansion code can favor efficiency over clarity.

It's in compiled code that efficiency counts most, and in compiled code the macro calls have already been expanded. If the expander code was efficient, it made compilation go slightly faster, but it won't make any difference in how well the program runs. Since the expansion of macro calls tends to be only a small part of the work done by a compiler, macros which expand efficiently can't usually make much of a difference even in the compilation speed. So most of the time you can safely write expander code the way you would write a quick, first version of a program. If the expander code does unnecessary work or conses a lot, so what? Your time is better spent improving other parts of the program. Certainly if there's a choice between clarity and speed in expander code, clarity should prevail. Macro definitions are generally harder to read than function definitions, because they contain a mix of expressions evaluated at two different times. If this confusion can be reduced at the expense of efficiency in the expander code, it's a bargain.

For example, suppose that we wanted to define a version of `and` as a macro. Since `(and a b c)` is equivalent to `(if a (if b c))`, we can write `and` in terms of `if` as in the first definition in Figure 7.9. According to the standards by which we judge ordinary code, `our-and` is badly written. The expander code is recursive, and on each recursion finds the length of successive cdrs of the same list. If this code were going to be evaluated at runtime, it would be better to define this macro as in `our-andb`, which generates the same expansion with no wasted effort. However, as a macro definition `our-and` is just as good, if not better. It may be inefficient in calling `length` on each recursion, but its organization shows more clearly the way in which the expansion depends on the number of conjuncts.

```
(defmacro our-and (&rest args)
  (case (length args)
    (0 t)
    (1 (car args))
    (t `(if ,(car args)
            (our-and ,@(cdr args))))))

(defmacro our-andb (&rest args)
  (if (null args)
      t
      (labels ((expander (rest)
                 (if (cdr rest)
                     `(if ,(car rest)
                          ,(expander (cdr rest)))
                     (car rest))))
        (expander args))))
```

Figure 7.9: Two macros equivalent to `and`.

As always, there are exceptions. In Lisp, the distinction between compile-time and runtime is an artificial one, so any rule which depends upon it is likewise artificial. In some programs, compile-time is runtime. If you're writing a program whose main purpose is transformation and which uses macros to do it, then everything changes: the expander code becomes your program, and the expansion its output. Of course under such circumstances expander code should be written with efficiency in mind. However, it's safe to say that most expander code (a) only affects the speed of compilation, and (b) doesn't affect it very much—meaning that clarity should nearly always come first.

With expansion code, it's just the opposite. Clarity matters less for macro expansions because they are rarely looked at, especially by other people. The forbidden goto is not entirely forbidden in expansions, and the disparaged `setq` not quite so disparaged.

Proponents of structured programming disliked goto for what it did to *source* code. It was not machine language jump instructions that they considered harmful—so long as they were hidden by more abstract constructs in source code. Gotos are condemned in Lisp precisely because it's so easy to hide them: you can use `do` instead, and if you didn't have `do`, you could write it. Of course, if we're going to build new abstractions on top of goto, the goto is going to have to exist somewhere. Thus it is not necessarily bad style to use `go` in the definition of a new macro, if it can't be written in terms of some existing macro.

Similarly, `setq` is frowned upon because it makes it hard to see where a given variable gets its value. However, a macroexpansion is not going to be read by many people, so there is usually little harm in using `setq` on variables created within the macroexpansion. If you look at expansions of some of the built-in macros, you'll see quite a lot of `setq`s.

Several circumstances can make clarity more important in expansion code. If you're writing a complicated macro, you may end up reading the expansions after all, at least while you're debugging it. Also, in simple macros, only a backquote separates expander code from expansion code, so if such macros generate ugly expansions, the ugliness will be all too visible in your source code. However, even when the clarity of expansion code becomes an issue, efficiency should still predominate. Efficiency is important in most runtime code. Two things make it especially so for macro expansions: their ubiquity and their invisibility.

Macros are often used to implement general-purpose utilities, which are then called everywhere in a program. Something used so often can't afford to be inefficient. What looks like a harmless little macro could, after the expansion of all the calls to it, amount to a significant proportion of your program. Such a macro should receive more attention than its length would seem to demand. Avoid consing especially. A utility which conses unnecessarily can ruin the performance of an otherwise efficient program.

The other reason to look to the efficiency of expansion code is its very invisibility. If a function is badly implemented, it will proclaim this fact to you every time you look at its definition. Not so with macros. From a macro definition, inefficiency in the expansion code may not be evident, which is all the more reason to go looking for it.

7.9 Dependence on Macros

If you redefine a function, other functions which call it will automatically get the new version.[6] The same doesn't always hold for macros. A macro call which occurs in a function definition gets replaced by its expansion when the function is compiled. What if we redefine the macro after the calling function has been compiled? Since no trace of the original macro call remains, the expansion within the function can't be updated. The behavior of the function will continue to reflect the *old* macro definition:

```
> (defmacro mac (x) `(1+ ,x))
MAC
```

[6]Except functions compiled inline, which impose the same restrictions on redefinition as macros.

```
> (setq fn (compile nil '(lambda (y) (mac y))))
#<Compiled-Function BF7E7E>
> (defmacro mac (x) `(+ ,x 100))
MAC
> (funcall fn 1)
2
```

Similar problems occur if code which calls some macro is compiled before the macro itself is defined. CLTL2 says that "a macro definition must be seen by the compiler before the first use of the macro." Implementations vary in how they respond to violations of this rule. Fortunately it's easy to avoid both types of problem. If you adhere to the following two principles, you need never worry about stale or nonexistent macro definitions:

1. Define macros before functions (or macros) which call them.
2. When a macro is redefined, also recompile all the functions (or macros) which call it—directly or via other macros.

It has been suggested that all the macros in a program be put in a separate file, to make it easier to ensure that macro definitions are compiled first. That's taking things too far. It would be reasonable to put general-purpose macros like `while` into a separate file, but general-purpose utilities ought to be separated from the rest of a program anyway, whether they're functions or macros.

Some macros are written just for use in one specific part of a program, and these should be defined with the code which uses them. So long as the definition of each macro appears before any calls to it, your programs will compile fine. Collecting together all your macros, simply because they're macros, would do nothing but make your code harder to read.

7.10 Macros from Functions

This section describes how to transform functions into macros. The first step in translating a function into a macro is to ask yourself if you really need to do it. Couldn't you just as well declare the function `inline` (p. 26)?

There are some legitimate reasons to consider how to translate functions into macros, though. When you begin writing macros, it sometimes helps to think as if you were writing a function—an approach that usually yields macros which aren't quite right, but which at least give you something to work from. Another reason to look at the relationship between macros and functions is to see how they *differ*. Finally, Lisp programmers sometimes actually want to convert functions into macros.

The difficulty of translating a function into a macro depends on a number of properties of the function. The easiest class to translate are the functions which

1. Have a body consisting of a single expression.
2. Have a parameter list consisting only of parameter names.
3. Create no new variables (except the parameters).
4. Are not recursive (nor part of a mutually recursive group).
5. Have no parameter which occurs more than once in the body.
6. Have no parameter whose value is used before that of another parameter occurring before it in the parameter list.
7. Contain no free variables.

One function which meets these criteria is the built-in Common Lisp function `second`, which returns the second element of a list. It could be defined:

```
(defun second (x) (cadr x))
```

Where a function definition meets all the conditions above, you can easily transform it into an equivalent macro definition. Simply put a backquote in front of the body and a comma in front of each symbol which occurs in the parameter list:

```
(defmacro second (x) `(cadr ,x))
```

Of course, the macro can't be called under all the same conditions. It can't be given as the first argument to `apply` or `funcall`, and it should not be called in environments where the functions it calls have new local bindings. For ordinary in-line calls, though, the macro `second` should do the same thing as the function `second`.

The technique changes slightly when the body has more than one expression, because a macro must expand into a single expression. So if condition 1 doesn't hold, you have to add a `progn`. The function `noisy-second`:

```
(defun noisy-second (x)
  (princ "Someone is taking a cadr!")
  (cadr x))
```

could be duplicated by the following macro:

```
(defmacro noisy-second (x)
  `(progn
     (princ "Someone is taking a cadr!")
     (cadr ,x)))
```

When the function doesn't meet condition 2 because it has an &rest or &body parameter, the rules are the same, except that the parameter, instead of simply having a comma before it, must be spliced into a call to list. Thus

```
(defun sum (&rest args)
  (apply #'+ args))
```

becomes

```
(defmacro sum (&rest args)
  `(apply #'+ (list ,@args)))
```

which in this case would be better rewritten:

```
(defmacro sum (&rest args)
  `(+ ,@args))
```

When condition 3 doesn't hold—when new variables are created within the function body—the rule about the insertion of commas must be modified. Instead of putting commas before all symbols in the parameter list, we only put them before those which will refer to the parameters. For example, in:

```
(defun foo (x y z)
  (list x (let ((x y))
            (list x z))))
```

neither of the last two instances of x will refer to the parameter x. The second instance is not evaluated at all, and the third instance refers to a new variable established by the let. So only the first instance will get a comma:

```
(defmacro foo (x y z)
  `(list ,x (let ((x ,y))
              (list x ,z))))
```

Functions can sometimes be transformed into macros when conditions 4, 5 and 6 don't hold. However, these topics are treated separately in later chapters. The issue of recursion in macros is covered in Section 10.4, and the dangers of multiple and misordered evaluation in Sections 10.1 and 10.2, respectively.

As for condition 7, it is possible to simulate closures with macros, using a technique similar to the error described on page 37. But seeing as this is a low hack, not consonant with the genteel tone of this book, we shall not go into details.

7.11 Symbol Macros

CLTL2 introduced a new kind of macro into Common Lisp, the symbol-macro. While a normal macro call looks like a function call, a symbol-macro "call" looks like a symbol.

Symbol-macros can only be locally defined. The `symbol-macrolet` special form can, within its body, cause a lone symbol to behave like an expression:

```
> (symbol-macrolet ((hi (progn (print "Howdy")
                               1)))
    (+ hi 2))
"Howdy"
3
```

The body of the `symbol-macrolet` will be evaluated as if every `hi` in argument position had been replaced with `(progn (print "Howdy") 1)`.

Conceptually, symbol-macros are like macros that don't take any arguments. With no arguments, macros become simply textual abbreviations. This is not to say that symbol-macros are useless, however. They are used in Chapter 15 (page 205) and Chapter 18 (page 237), and in the latter instance they are indispensable.

8

When to Use Macros

How do we know whether a given function should really be a function, rather than a macro? Most of the time there is a clear distinction between the cases which call for macros and those which don't. By default we should use functions: it is inelegant to use a macro where a function would do. We should use macros only where they bring us some specific advantage.

When do macros bring advantages? That is the subject of this chapter. Usually the question is not one of advantage, but necessity. Most of the things we do with macros, we could not do with functions. Section 8.1 lists the kinds of operators ◦ which can only be implemented as macros. However, there is also a small (but interesting) class of borderline cases, in which an operator might justifiably be written as a function or a macro. For these situations, Section 8.2 gives the arguments for and against macros. Finally, having considered what macros are capable of doing, we turn in Section 8.3 to a related question: what kinds of things do people do with them?

8.1 When Nothing Else Will Do

It's a general principle of good design that if you find similar code appearing at several points in a program, you should write a subroutine and replace the similar sequences of code with calls to the subroutine. When we apply this principle to Lisp programs, we have to decide whether the "subroutine" should be a function or a macro.

In some cases it's easy to decide to write a macro instead of a function, because only a macro can do what's needed. A function like `1+` could conceivably be written as either a function or a macro:

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

But `while`, from Section 7.3, could only be defined as a macro:

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

There is no way to duplicate the behavior of this macro with a function. The definition of `while` splices the expressions passed as `body` into the body of a `do`, where they will be evaluated only if the `test` expression returns `nil`. No function could do that; in a function call, all the arguments are evaluated before the function is even invoked.

When you do need a macro, what do you need from it? Macros can do two things that functions can't: they can control (or prevent) the evaluation of their arguments, and they are expanded right into the calling context. Any application which requires macros requires, in the end, one or both of these properties.

The informal explanation that "macros don't evaluate their arguments" is slightly wrong. It would be more precise to say that macros *control* the evaluation of the arguments in the macro call. Depending on where the argument is placed in the macro's expansion, it could be evaluated once, many times, or not at all. Macros use this control in four major ways:

1. *Transformation.* The Common Lisp `setf` macro is one of a class of macros which pick apart their arguments before evaluation. A built-in access function will often have a converse whose purpose is to set what the access function retrieves. The converse of `car` is `rplaca`, of `cdr`, `rplacd`, and so on. With `setf` we can use calls to such access functions as if they were variables to be set, as in `(setf (car x) 'a)`, which could expand into `(progn (rplaca x 'a) 'a)`.

   To perform this trick, `setf` has to look inside its first argument. To know that the case above requires `rplaca`, `setf` must be able to see that the first argument is an expression beginning with `car`. Thus `setf`, and any other operator which transforms its arguments, must be written as a macro.

2. *Binding.* Lexical variables must appear directly in the source code. The first argument to `setq` is not evaluated, for example, so anything built on `setq` must be a macro which expands into a `setq`, rather than a function which

calls it. Likewise for operators like `let`, whose arguments are to appear as parameters in a `lambda` expression, for macros like `do` which expand into `let`s, and so on. Any new operator which is to alter the lexical bindings of its arguments must be written as a macro.

3. *Conditional evaluation.* All the arguments to a function are evaluated. In constructs like `when`, we want some arguments to be evaluated only under certain conditions. Such flexibility is only possible with macros.

4. *Multiple evaluation.* Not only are the arguments to a function all evaluated, they are all evaluated exactly once. We need a macro to define a construct like `do`, where certain arguments are to be evaluated repeatedly.

There are also several ways to take advantage of the inline expansion of macros. It's important to emphasize that the expansions thus appear in the lexical context of the macro call, since two of the three uses for macros depend on that fact. They are:

5. *Using the calling environment.* A macro can generate an expansion containing a variable whose binding comes from the context of the macro call. The behavior of the following macro:

   ```
   (defmacro foo (x)
     `(+ ,x y))
   ```

   depends on the binding of `y` where `foo` is called.

   This kind of lexical intercourse is usually viewed more as a source of contagion than a source of pleasure. Usually it would be bad style to write such a macro. The ideal of functional programming applies as well to macros: the preferred way to communicate with a macro is through its parameters. Indeed, it is so rarely necessary to use the calling environment that most of the time it happens, it happens by mistake. (See Chapter 9.) Of all the macros in this book, only the continuation-passing macros (Chapter 20) and some parts of the ATN compiler (Chapter 23) use the calling environment in this way.

6. *Wrapping a new environment.* A macro can also cause its arguments to be evaluated in a new lexical environment. The classic example is `let`, which could be implemented as a macro on `lambda` (page 144). Within the body of an expression like `(let ((y 2)) (+ x y))`, `y` will refer to a new variable.

7. *Saving function calls.* The third consequence of the inline insertion of macroexpansions is that in compiled code there is no overhead associated with a macro call. By runtime, the macro call has been replaced by its expansion. (The same is true in principle of functions declared `inline`.)

Significantly, cases 5 and 6, when unintentional, constitute the problem of variable capture, which is probably the worst thing a macro writer has to fear. Variable capture is discussed in Chapter 9.

Instead of seven ways of using macros, it might be better to say that there are six and a half. In an ideal world, all Common Lisp compilers would obey `inline` declarations, and saving function calls would be a task for inline functions, not macros. An ideal world is left as an exercise to the reader.

8.2 Macro or Function?

The previous section dealt with the easy cases. Any operator that needs access to its parameters before they are evaluated should be written as a macro, because there is no other choice. What about those operators which could be written either way? Consider for example the operator `avg`, which returns the average of its arguments. It could be defined as a function

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

but there is a good case for defining it as a macro,

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

because the function version would entail an unnecessary call to `length` each time `avg` was called. At compile-time we may not know the values of the arguments, but we do know how many there are, so the call to `length` could just as well be made then. Here are several points to consider when we face such choices:

THE PROS

1. *Computation at compile-time.* A macro call involves computation at two times: when the macro is expanded, and when the expansion is evaluated. All the macroexpansion in a Lisp program is done when the program is compiled, and every bit of computation which can be done at compile-time is one bit that won't slow the program down when it's running. If an operator could be written to do some of its work in the macroexpansion stage, it will be more efficient to make it a macro, because whatever work a

smart compiler can't do itself, a function has to do at runtime. Chapter 13 describes macros like `avg` which do some of their work during the expansion phase.

2. *Integration with Lisp.* Sometimes, using macros instead of functions will make a program more closely integrated with Lisp. Instead of writing a program to solve a certain problem, you may be able to use macros to transform the problem into one that Lisp already knows how to solve. This approach, when possible, will usually make programs both smaller and more efficient: smaller because Lisp is doing some of your work for you, and more efficient because production Lisp systems generally have had more of the fat sweated out of them than user programs. This advantage appears mostly in embedded languages, which are described starting in Chapter 19.

3. *Saving function calls.* A macro call is expanded right into the code where it appears. So if you write some frequently used piece of code as a macro, you can save a function call every time it's used. In earlier dialects of Lisp, programmers took advantage of this property of macros to save function calls at runtime. In Common Lisp, this job is supposed to be taken over by functions declared `inline`.

   By declaring a function to be `inline`, you ask for it to be compiled right into the calling code, just like a macro. However, there is a gap between theory and practice here; CLTL2 (p. 229) says that "a compiler is free to ignore this declaration," and some Common Lisp compilers do. It may still be reasonable to use macros to save function calls, if you are compelled to use such a compiler.

In some cases, the combined advantages of efficiency and close integration with Lisp can create a strong argument for the use of macros. In the query compiler of Chapter 19, the amount of computation which can be shifted forward to compile-time is so great that it justifies turning the whole program into a single giant macro. Though done for speed, this shift also brings the program closer to Lisp: in the new version, it's easier to use Lisp expressions—arithmetic expressions, for example—within queries.

THE CONS

4. *Functions are data,* while macros are more like instructions to the compiler. Functions can be passed as arguments (e.g. to `apply`), returned by functions, or stored in data structures. None of these things are possible with macros.

   In some cases, you can get what you want by enclosing the macro call within a lambda-expression. This works, for example, if you want to `apply` or `funcall` certain macros:

```
> (funcall #'(lambda (x y) (avg x y)) 1 3)
2
```

However, this is an inconvenience. It doesn't always work, either: even if, like `avg`, the macro has an `&rest` parameter, there is no way to pass it a varying number of arguments.

5. *Clarity of source code.* Macro definitions can be harder to read than the equivalent function definitions. So if writing something as a macro would only make a program marginally better, it might be better to use a function instead.

6. *Clarity at runtime.* Macros are sometimes harder to debug than functions. If you get a runtime error in code which contains a lot of macro calls, the code you see in the backtrace could consist of the expansions of all those macro calls, and may bear little resemblance to the code you originally wrote.

   And because macros disappear when expanded, they are not accountable at runtime. You can't usually use `trace` to see how a macro is being called. If it worked at all, `trace` would show you the call to the macro's expander function, not the macro call itself.

7. *Recursion.* Using recursion in macros is not so simple as it is in functions. Although the expansion function of a macro may be recursive, the expansion itself may not be. Section 10.4 deals with the subject of recursion in macros.

All these considerations have to be balanced against one another in deciding when to use macros. Only experience can tell which will predominate. However, the examples of macros which appear in later chapters cover most of the situations in which macros are useful. If a potential macro is analogous to one given here, then it is probably safe to write it as such.

Finally, it should be noted that clarity at runtime (point 6) rarely becomes an issue. Debugging code which uses a lot of macros will not be as difficult as you might expect. If macro definitions were several hundred lines long, it might be unpleasant to debug their expansions at runtime. But utilities, at least, tend to be written in small, trusted layers. Generally their definitions are less than 15 lines long. So even if you are reduced to poring over backtraces, such macros will not cloud your view very much.

8.3 Applications for Macros

Having considered what can be done with macros, the next question to ask is: in what sorts of applications can we use them? The closest thing to a general description of macro use would be to say that they are used mainly for syntactic transformations. This is not to suggest that the scope for macros is restricted. Since Lisp programs are made from[1] lists, which are Lisp data structures, "syntactic transformation" can go a long way indeed. Chapters 19–24 present whole programs whose purpose could be described as syntactic transformation, and which are, in effect, all macro.

Macro applications form a continuum between small general-purpose macros like `while`, and the large, special-purpose macros defined in the later chapters. On one end are the *utilities*, the macros resembling those that every Lisp has built-in. They are usually small, general, and written in isolation. However, you can write utilities for specific classes of programs too, and when you have a collection of macros for use in, say, graphics programs, they begin to look like a programming language for graphics. At the far end of the continuum, macros allow you to write whole programs in a language distinctly different from Lisp. Macros used in this way are said to implement *embedded languages.*

Utilities are the first offspring of the bottom-up style. Even when a program is too small to be built in layers, it may still benefit from additions to the lowest layer, Lisp itself. The utility `nil!`, which sets its argument to `nil`, could not be defined except as a macro:

```
(defmacro nil! (x)
  `(setf ,x nil))
```

Looking at `nil!`, one is tempted to say that it doesn't *do* anything, that it merely saves typing. True, but all any macro does is save typing. If one wants to think of it in these terms, the job of a compiler is to save typing in machine language. The value of utilities should not be underestimated, because their effect is cumulative: several layers of simple macros can make the difference between an elegant program and an incomprehensible one.

Most utilities are patterns embodied. When you notice a pattern in your code, consider turning it into a utility. Patterns are just the sort of thing computers are good at. Why should you bother following them when you could have a program do it for you? Suppose that in writing some program you find yourself using in many different places `do` loops of the same general form:

```
(do ()
    ((not ⟨condition⟩))
  . ⟨body of code⟩)
```

[1] *Made from,* in the sense that lists are the input to the compiler. Functions are no longer *made of* lists, as they used to be in some earlier dialects.

When you find a pattern repeated through your code, that pattern often has a name. The name of this pattern is *while*. If we want to provide it in a new utility, we will have to use a macro, because we need conditional and repeated evaluation. If we define `while` using this definition from page 91:

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

then we can replace the instances of the pattern with

```
(while ⟨condition⟩
   .  ⟨body of code⟩)
```

Doing so will make the code shorter and also make it declare in a clearer voice what it's doing.

The ability to transform their arguments makes macros useful in writing interfaces. The appropriate macro will make it possible to type a shorter, simpler expression where a long or complex one would have been required. Although graphic interfaces decrease the need to write such macros for end users, programmers use this type of macro as much as ever. The most common example is `defun`, which makes the binding of functions resemble, on the surface, a function definition in a language like Pascal or C. Chapter 2 mentioned that the following two expressions have approximately the same effect:

```
(defun foo (x) (* x 2))

(setf (symbol-function 'foo)
      #'(lambda (x) (* x 2)))
```

Thus `defun` can be implemented as a macro which turns the former into the latter. We could imagine it written as follows:[2]

```
(defmacro our-defun (name parms &body body)
  `(progn
     (setf (symbol-function ',name)
           #'(lambda ,parms (block ,name ,@body)))
     ',name))
```

Macros like `while` and `nil!` could be described as general-purpose utilities. Any Lisp program might use them. But particular domains can have their utilities

[2]For clarity, this version ignores all the bookkeeping that `defun` must perform.

```
(defun move-objs (objs dx dy)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (incf (obj-x o) dx)
      (incf (obj-y o) dy))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))

(defun scale-objs (objs factor)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (setf (obj-dx o) (* (obj-dx o) factor)
            (obj-dy o) (* (obj-dy o) factor)))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))
```

Figure 8.1: Original move and scale.

as well. There is no reason to suppose that base Lisp is the only level at which you have a programming language to extend. If you're writing a CAD program, for example, the best results will sometimes come from writing it in two layers: a language (or if you prefer a more modest term, a toolkit) for CAD programs, and in the layer above, your particular application.

Lisp blurs many distinctions which other languages take for granted. In other languages, there really are conceptual distinctions between compile-time and runtime, program and data, language and program. In Lisp, these distinctions exist only as conversational conventions. There is no line dividing, for example, language and program. You can draw the line wherever suits the problem at hand. So it really is no more than a question of terminology whether to call an underlying layer of code a toolkit or a language. One advantage of considering it as a language is that it suggests you can extend this language, as you do Lisp, with utilities.

Suppose we are writing an interactive 2D drawing program. For simplicity, we will assume that the only objects handled by the program are line segments, represented as an origin $\langle x,y \rangle$ and a vector $\langle dx,dy \rangle$. One of the things such a program will have to do is slide groups of objects. This is the purpose of the function `move-objs` in Figure 8.1. For efficiency, we don't want to redraw the whole screen after each operation—only the parts which have changed. Hence

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    `(let ((,gob ,objs))
       (multiple-value-bind (,x0 ,y0 ,x1 ,y1) (bounds ,gob)
         (dolist (,var ,gob) ,@body)
         (multiple-value-bind (xa ya xb yb) (bounds ,gob)
           (redraw (min ,x0 xa) (min ,y0 ya)
                   (max ,x1 xb) (max ,y1 yb)))))))

(defun move-objs (objs dx dy)
  (with-redraw (o objs)
    (incf (obj-x o) dx)
    (incf (obj-y o) dy)))

(defun scale-objs (objs factor)
  (with-redraw (o objs)
    (setf (obj-dx o) (* (obj-dx o) factor)
          (obj-dy o) (* (obj-dy o) factor))))
```

Figure 8.2: Move and scale filleted.

the two calls to the function `bounds`, which returns four coordinates (min x, min y, max x, max y) representing the bounding rectangle of a group of objects. The operative part of `move-objs` is sandwiched between two calls to `bounds` which find the bounding rectangle before and then after the movement, and then redraw the entire affected region.

The function `scale-objs` is for changing the size of a group of objects. Since the bounding region could grow or shrink depending on the scale factor, this function too must do its work between two calls to `bounds`. As we wrote more of the program, we would see more of this pattern: in functions to rotate, flip, transpose, and so on.

With a macro we can abstract out the code that these functions would all have in common. The macro `with-redraw` in Figure 8.2 provides the skeleton that the functions in Figure 8.1 share.[3] As a result, they can now be defined in four lines each, as at the end of Figure 8.2. With these two functions the new macro has already paid for itself in brevity. And how much clearer the two functions ○

[3]The definition of this macro anticipates the next chapter by using gensyms. Their purpose will be explained shortly.

become once the details of screen redrawing are abstracted away.

One way to view `with-redraw` is as a construct in a language for writing interactive drawing programs. As we develop more such macros, they will come to resemble a programming language in fact as well as in name, and our application itself will begin to show the elegance one would expect in a program written in a language defined for its specific needs.

The other major use of macros is to implement embedded languages. Lisp is an exceptionally good language in which to write programming languages, because Lisp programs can be expressed as lists, and Lisp has a built-in parser (`read`) and compiler (`compile`) for programs so expressed. Most of the time you don't even have to call `compile`; you can have your embedded language compiled implicitly, by compiling the code which does the transformations (page 25).

An embedded language is one which is not written on top of Lisp so much as commingled with it, so that the syntax is a mixture of Lisp and constructs specific to the new language. The naive way to implement an embedded language is to write an interpreter for it in Lisp. A better approach, when possible, is to implement the language by transformation: transform each expression into the Lisp code that the interpreter would have run in order to evaluate it. That's where macros come in. The job of macros is precisely to transform one type of expression into another, so they're the natural choice when writing embedded languages.

In general, the more an embedded language can be implemented by transformation, the better. For one, it's less work. If the new language has arithmetic, for example, you needn't face all the complexities of representing and manipulating numeric quantities. If Lisp's arithmetic capabilities are sufficient for your purposes, then you can simply transform your arithmetic expressions into the equivalent Lisp ones, and leave the rest to the Lisp.

Using transformation will ordinarily make your embedded languages faster as well. Interpreters have inherent disadvantages with respect to speed. When code occurs within a loop, for example, an interpreter will often have to do work on each iteration which in compiled code could have been done just once. An embedded language which has its own interpreter will therefore be slow, even if the interpreter itself is compiled. But if the expressions in the new language are transformed into Lisp, the resulting code can then be compiled by the Lisp compiler. A language so implemented need suffer none of the overheads of interpretation at runtime. Short of writing a true compiler for your language, macros will yield the best performance. In fact, the macros which transform the new language can be seen as a compiler for it—just one which relies on the existing Lisp compiler to do most of the work.

We won't consider any examples of embedded languages here, since Chapters 19–25 are all devoted to the topic. Chapter 19 deals specifically with the difference between interpreting and transforming embedded languages, and shows the same language implemented by each of the two methods.

One book on Common Lisp asserts that the scope for macros is limited, citing as evidence the fact that, of the operators defined in CLTL1, less than 10% were macros. This is like saying that since our house is made of bricks, our furniture will be too. The proportion of macros in a Common Lisp program will depend entirely on what it's supposed to do. Some programs will contain no macros. Some programs could be all macros.

9

Variable Capture

Macros are vulnerable to a problem called *variable capture*. Variable capture occurs when macroexpansion causes a name clash: when some symbol ends up referring to a variable from another context. Inadvertent variable capture can cause extremely subtle bugs. This chapter is about how to foresee and avoid them. However, intentional variable capture is a useful programming technique, and Chapter 14 is full of macros which rely on it.

9.1 Macro Argument Capture

A macro vulnerable to unintended variable capture is a macro with a bug. To avoid writing such macros, we must know precisely when capture can occur. Instances of variable capture can be traced to one of two situations: macro argument capture and free symbol capture. In argument capture, a symbol passed as an argument in the macro call inadvertently refers to a variable established by the macro expansion itself. Consider the following definition of the macro `for`, which iterates over a body of expressions like a Pascal `for` loop:

```
(defmacro for ((var start stop) &body body)          ; wrong
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

This macro looks correct at first sight. It even seems to work fine:

```
> (for (x 1 5)
    (princ x))
12345
NIL
```

Indeed, the error is so subtle that we might use this version of the macro hundreds of times and have it always work perfectly. Not if we call it this way, though:

```
(for (limit 1 5)
  (princ limit))
```

We might expect this expression to have the same effect as the one before. But it doesn't print anything; it generates an error. To see why, we look at its expansion:

```
(do ((limit 1 (1+ limit))
     (limit 5))
    ((> limit limit))
  (princ limit))
```

Now it's obvious what goes wrong. There is a name clash between a symbol local to the macro expansion and a symbol passed as an argument to the macro. The macroexpansion *captures* `limit`. It ends up occurring twice in the same `do`, which is illegal.

Errors caused by variable capture are rare, but what they lack in frequency they make up in viciousness. This capture was comparatively mild—here, at least, we got an error. More often than not, a capturing macro would simply yield incorrect results with no indication that anything was wrong. In this case,

```
> (let ((limit 5))
    (for (i 1 10)
      (when (> i limit)
        (princ i))))
NIL
```

the resulting code quietly does nothing.

9.2 Free Symbol Capture

Less frequently, the macro definition itself contains a symbol which inadvertently refers to a binding in the environment where the macro is expanded. Suppose some program, instead of printing warnings to the user as they arise, wants to store the warnings in a list, to be examined later. One person writes a macro `gripe`, which takes a warning message and adds it to a global list, `w`:

```
(defvar w nil)

(defmacro gripe (warning)                             ; wrong
  `(progn (setq w (nconc w (list ,warning)))
          nil))
```

Someone else then wants to write a function `sample-ratio`, to return the ratio of the lengths of two lists. If either of the lists has less than two elements, the function is to return `nil` instead, also issuing a warning that it was called on a statistically insignificant case. (Actual warnings could be more informative, but their content isn't relevant to this example.)

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
        (gripe "sample < 2")
        (/ vn wn))))
```

If sample-ratio is called with `w = (b)`, then it will want to warn that one of its arguments, with only one element, is statistically insignificant. But when the call to `gripe` is expanded, it will be as if `sample-ratio` had been defined:

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
        (progn (setq w (nconc w (list "sample < 2")))
               nil)
        (/ vn wn))))
```

The problem here is that `gripe` is used in a context where `w` has its own local binding. The warning, instead of being saved in the global warning list, will be nconced onto the end of one of the parameters of `sample-ratio`. Not only is the warning lost, but the list `(b)`, which is probably used as data elsewhere in the program, will have an extraneous string appended to it:

```
> (let ((lst '(b)))
    (sample-ratio nil lst)
    lst)
(B "sample < 2")
> w
NIL
```

9.3 When Capture Occurs

It's asking a lot of the macro writer to be able to look at a macro definition and foresee all the possible problems arising from these two types of capture. Variable capture is a subtle matter, and it takes some experience to anticipate all the ways a capturable symbol could wreak mischief in a program. Fortunately, you can detect and eliminate capturable symbols in your macro definitions without having to think about *how* their capture could send your program awry. This section provides a straightforward rule for detecting capturable symbols. The remaining sections of this chapter explain techniques for eliminating them.

The rule for defining a capturable variable depends on some subordinate concepts, which must be defined first:

Free: A symbol *s* occurs free in an expression when it is used as a variable in that expression, but the expression does not create a binding for it.

In the following expression,

```
(let ((x y) (z 10))
  (list w x z))
```

`w`, `x` and `z` all occur free within the `list` expression, which establishes no bindings. However, the enclosing `let` expression establishes bindings for `x` and `z`, so within the `let` as a whole, only `y` and `w` occur free. Note that in

```
(let ((x x))
  x)
```

the second instance of `x` is free—it's not within the scope of the new binding being established for `x`.

Skeleton: The skeleton of a macro expansion is the whole expansion, minus anything which was part of an argument in the macro call.

If `foo` is defined:

```
(defmacro foo (x y)
  `(/ (+ ,x 1) ,y))
```

and called thus:

```
(foo (- 5 2) 6)
```

then it yields the macro expansion:

```
(/ (+ (- 5 2) 1) 6)
```

The skeleton of this expansion is the above expression with holes where the parameters x and y got inserted:

```
(/ (+         1)  )
```

With these two concepts defined, it's possible to state a concise rule for detecting capturable symbols:

Capturable: A symbol is capturable in some macro expansion if (a) it occurs free in the skeleton of the macro expansion, or (b) it is bound by a part of the skeleton in which arguments passed to the macro are either bound or evaluated.

Some examples will show the implications of this rule. In the simplest case:

```
(defmacro cap1 ()
  '(+ x 1))
```

x is capturable because it will occur free in the skeleton. That's what caused the bug in gripe. In this macro:

```
(defmacro cap2 (var)
  `(let ((x ...)
         (,var ...))
     ...))
```

x is capturable because it is bound in an expression where an argument to the macro call will also be bound. (That's what went wrong in for.) Likewise for the following two macros

```
(defmacro cap3 (var)
  `(let ((x ...))
     (let ((,var ...))
       ...)))

(defmacro cap4 (var)
  `(let ((,var ...))
     (let ((x ...))
       ...)))
```

in both of which x is capturable. However, if there is no context in which the binding of x and the variable passed as an argument will both be visible, as in

```
(defmacro safe1 (var)
  `(progn (let ((x 1))
            (print x))
          (let ((,var 1))
            (print ,var))))
```

then x won't be capturable. Not all variables bound by the skeleton are at risk. However, if arguments to the macro call are evaluated within a binding established by the skeleton,

```
(defmacro cap5 (&body body)
  `(let ((x ...))
     ,@body))
```

then variables so bound are at risk of capture: in `cap5`, x is capturable. In this case, though,

```
(defmacro safe2 (expr)
  `(let ((x ,expr))
     (cons x 1)))
```

x is *not* capturable, because when the argument passed to `expr` is evaluated, the new binding of x won't be visible. Note also that it's only the binding of skeletal variables we have to worry about. In this macro

```
(defmacro safe3 (var &body body)
  `(let ((,var ...))
     ,@body))
```

no symbol is at risk of inadvertent capture (assuming that the user expects that the first argument will be bound).

Now let's look at the original definition of `for` in light of the new rule for identifying capturable symbols:

```
(defmacro for ((var start stop) &body body)          ; wrong
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

It turns out now that this definition of `for` is vulnerable to capture in *two* ways: `limit` could be passed as the first argument to `for`, as in the original example:

```
(for (limit 1 5)
  (princ limit))
```

but it's just as dangerous if `limit` occurs in the body of the loop:

```
(let ((limit 0))
  (for (x 1 10)
    (incf limit x))
  limit)
```

Someone using `for` in this way would be expecting his own binding of `limit` to be the one incremented in the loop, and the expression as a whole to return 55; in fact, only the binding of `limit` generated by the skeleton of the expansion will be incremented:

```
(do ((x 1 (1+ x))
     (limit 10))
    ((> x limit))
  (incf limit x))
```

and since that's the one which controls iteration, the loop won't even terminate.

The rules presented in this section should be used with the reservation that they are intended only as a guide. They are not even formally stated, let alone formally correct. The problem of capture is a vaguely defined one, since it depends on expectations. For example, in an expression like

```
(let ((x 1)) (list x))
```

we don't regard it as an error that when `(list x)` is evaluated, `x` will refer to a new variable. That's what `let` is supposed to do. The rules for detecting capture are also imprecise. You could write macros which passed these tests, and which still would be vulnerable to unintended capture. For example,

```
(defmacro pathological (&body body)                   ; wrong
  (let* ((syms (remove-if (complement #'symbolp)
                          (flatten body)))
         (var (nth (random (length syms))
                   syms)))
    `(let ((,var 99))
       ,@body)))
```

When this macro is called, the expressions in the body will be evaluated as if in a `progn`—but one random variable within the body may have a different value. This is clearly capture, but it passes our tests, because the variable does not occur in the skeleton. In practice, though, the rules will work nearly all the time: one rarely (if ever) wants to write a macro like the example above.

Vulnerable to capture:

```
(defmacro before (x y seq)
  `(let ((seq ,seq))
     (< (position ,x seq)
        (position ,y seq))))
```

A correct version:

```
(defmacro before (x y seq)
  `(let ((xval ,x) (yval ,y) (seq ,seq))
     (< (position xval seq)
        (position yval seq))))
```

Figure 9.1: Avoiding capture with `let`.

9.4 Avoiding Capture with Better Names

The first two sections divided instances of variable capture into two types: argument capture, where a symbol used in an argument is caught by a binding established by the macro skeleton, and free symbol capture, where a free symbol in a macroexpansion is caught by a binding in force where the macro is expanded. The latter cases are usually dealt with simply by giving global variables distinguished names. In Common Lisp, it is traditional to give global variables names which begin and end with asterisks. The variable defining the current package is called `*package*`, for example. (Such a name may be pronounced "star-package-star" to emphasize that it is not an ordinary variable.)

So really it was the responsibility of the author of `gripe` to store warnings in a variable called something like `*warnings*`, rather than just `w`. If the author of `sample-ratio` had used `*warnings*` as a parameter, then he would deserve every bug he got, but he can't be blamed for thinking that it would be safe to call a parameter `w`.

9.5 Avoiding Capture by Prior Evaluation

Sometimes argument capture can be cured simply by evaluating the endangered arguments outside of any bindings created by the macroexpansion. The simplest cases can be handled by beginning the macro with a `let` expression. Figure 9.1 contains two versions of the macro `before`, which takes two objects and a sequence, and returns true iff the first object occurs before the second in the sequence.[1] The first definition is incorrect. Its initial `let` ensures that the form passed as `seq` is only evaluated once, but it is not sufficient to avoid the following problem:

```
> (before (progn (setq seq '(b a)) 'a)
          'b
          '(a b))
NIL
```

This amounts to asking "Is `a` before `b` in `(a b)`?" If `before` were correct, it would return true. Macroexpansion shows what really happens: the evaluation of the first argument to `<` rearranges the list to be searched in the second.

```
(let ((seq '(a b)))
  (< (position (progn (setq seq '(b a)) 'a)
               seq)
     (position 'b seq)))
```

To avoid this problem, it will suffice to evaluate all the arguments first in one big `let`. The second definition in Figure 9.1 is thus safe from capture.

Unfortunately, the `let` technique works only in a narrow range of cases: macros where

1. all the arguments at risk of capture are evaluated exactly once, and
2. none of the arguments need to be evaluated in the scope of bindings established by the macro skeleton.

This rules out a great many macros. The proposed `for` macro violates both conditions. However, we can use a variation of this scheme to make macros like `for` safe from capture: to wrap its `body` forms within a lambda-expression outside of any locally created bindings.

Some macros, including those for iteration, yield expansions where expressions appearing in the macro call will be evaluated within newly established bindings. In the definition of `for`, for example, the body of the loop must be evaluated within a `do` created by the macro. Variables occurring in the body of the loop are thus vulnerable to capture by bindings established by the `do`. We can protect variables in the body from such capture by wrapping the body in a closure, and, within the loop, instead of inserting the expressions themselves, simply funcalling the closure.

○ Figure 9.2 shows a version of `for` which uses this technique. Since the closure

[1] This macro is used only as an example. Really it should neither be implemented as a macro, nor use the inefficient algorithm that it does. For a proper definition, see page 50.

Vulnerable to capture:

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

A correct version:

```
(defmacro for ((var start stop) &body body)
  `(do ((b #'(lambda (,var) ,@body))
        (count ,start (1+ count))
        (limit ,stop))
       ((> count limit))
     (funcall b count)))
```

Figure 9.2: Avoiding capture with a closure.

is the first thing made by the expansion of a `for`, free symbols occurring in the body will all refer to variables in the environment of the macro call. Now the `do` communicates with its body through the parameters of the closure. All the closure needs to know from the `do` is the number of the current iteration, so it has only one parameter, the symbol specified as the index variable in the macro call.

The technique of wrapping expressions in lambdas is not a universal remedy. You can use it to protect a body of code, but closures won't be any use when, for example, there is a risk of the same variable being bound twice by the same `let` or `do` (as in our original broken `for`). Fortunately, in this case, by rewriting `for` to package its body in a closure, we also eliminated the need for the `do` to establish bindings for the `var` argument. The `var` argument of the old `for` became the parameter of the closure and could be replaced in the `do` by an actual symbol, `count`. So the new definition of `for` is completely immune from capture, as the test in Section 9.3 will show.

The disadvantage of using closures is that they might be less efficient. We could be introducing another function call. Potentially worse, if the compiler doesn't give the closure dynamic extent, space for it will have to be allocated in the heap at runtime.

9.6 Avoiding Capture with Gensyms

There is one certain way to avoid macro argument capture: replacing capturable symbols with gensyms. In the original version of `for`, problems arise when two symbols inadvertently have the same name. If we want to avoid the possibility that a macro skeleton will contain a symbol also used by the calling code, we might hope to get away with using only strangely named symbols in macro definitions:

```
(defmacro for ((var start stop) &body body)          ; wrong
  `(do ((,var ,start (1+ ,var))
        (xsf2jsh ,stop))
       ((> ,var xsf2jsh))
     ,@body))
```

but this is no solution. It doesn't eliminate the bug, just makes it less likely to show. And not so very less likely at that—it's still possible to imagine conflicts arising in nested instances of the same macro.

We need some way to *ensure* that a symbol is unique. The Common Lisp ○ function `gensym` exists just for this purpose. It returns a symbol, called a *gensym,* which is guaranteed not to be `eq` to any symbol either typed in or constructed by a program.

How can Lisp promise this? In Common Lisp, each package keeps a list of all the symbols known in that package. (For an introduction to packages, see page 381.) A symbol which is on the list is said to be *interned* in the package. Each call to `gensym` returns a unique, uninterned symbol. And since every symbol seen by `read` gets interned, no one could type anything identical to a gensym. Thus, if you begin the expression

```
(eq (gensym) ...
```

there is no way to complete it that will cause it to return true.

Asking `gensym` to make you a symbol is like taking the approach of choosing a strangely named symbol one step further—`gensym` will give you a symbol whose name isn't even in the phone book. When Lisp has to display a gensym,

```
> (gensym)
#:G47
```

what it prints is really just Lisp's equivalent of "John Doe," an arbitrary name made up for something whose name is irrelevant. And to be sure that we don't have any illusions about this, gensyms are displayed preceded by a sharp-colon, a special read-macro which exists just to cause an error if we ever try to read the gensym in again.

Vulnerable to capture:

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

A correct version:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

Figure 9.3: Avoiding capture with `gensym`.

In CLTL2 Common Lisp, the number in a gensym's printed representation comes from `*gensym-counter*`, a global variable always bound to an integer. By resetting this counter we can cause two gensyms to print the same

```
> (setq x (gensym))
#:G48
> (setq *gensym-counter* 48 y (gensym))
#:G48
> (eq x y)
NIL
```

but they won't be identical.

Figure 9.3 contains a correct definition of `for` using gensyms. Now there is no `limit` to clash with symbols in forms passed to the macro. It has been replaced by a symbol gensymed on the spot. In each expansion of the macro, the place of `limit` will be taken by a unique symbol created at expansion-time.

The correct definition of `for` is a complicated one to produce on the first try. Finished code, like a finished theorem, often covers up a lot of trial and error. So don't worry if you have to write several versions of a macro. To begin writing macros like `for`, you may want to write the first version without thinking about variable capture, and then to go back and make gensyms for symbols which could be involved in captures.

9.7 Avoiding Capture with Packages

To some extent, it is possible to avoid capture by defining macros in their own package. If you create a `macros` package and define `for` there, you can even use the definition given first

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

and call it safely from any other package. If you call `for` from another package, say `mycode`, then even if you do use `limit` as the first argument, it will be `mycode::limit`—a distinct symbol from `macros::limit`, which occurs in the macro skeleton.

However, packages do not provide a very general solution to the problem of capture. In the first place, macros are an integral part of some programs, and it would be inconvenient to have to separate them in their own package. Second, this approach offers no protection against capture by other code in the `macros` package.

9.8 Capture in Other Name-Spaces

The previous sections have spoken of capture as if it were a problem which afflicted variables exclusively. Although most capture is variable capture, the problem can arise in Common Lisp's other name-spaces as well.

Functions may also be locally bound, and function bindings are equally liable to inadvertent capture. For example:

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
    (mac 10))
9
```

As predicted by the capture rule, the `fn` which occurs free in the skeleton of `mac` is at risk of capture. When `fn` is locally rebound, `mac` returns a different value than it does generally.

What to do about this case? When the symbol at risk of capture is the name of a built-in function or macro, then it's reasonable to do nothing. In CLTL2 (p. 260) ○ if the name of anything built-in is given a local function or macro binding, "the consequences are undefined." So it wouldn't matter what your macro did—anyone who rebinds built-in functions is going to have problems with more than just your macros.

Otherwise, you can protect function names against macro argument capture the same way you would protect variable names: by using gensyms as names for any functions given local definitions by the macro skeleton. Avoiding free symbol capture, as in the case above, is a bit more difficult. The way to protect variables against free symbol capture was to give them distinctly global names: e.g. `*warnings*` instead of `w`. This solution is not practical for functions, because there is no convention for distinguishing the names of global functions—most functions are global. If you're concerned about a macro being called in an environment where a function it needs might be locally redefined, the best solution is probably to put your code in a distinct package.

Block-names are also liable to capture, as are the tags used by `go` and `throw`. When your macros need such symbols, you should use gensyms, as in the definition of `our-do` on page 98.

Remember also that operators like `do` are implicitly enclosed in a block named `nil`. Thus a `return` or `return-from nil` within a `do` returns from the `do`, not the containing expression:

```
> (block nil
    (list 'a
          (do ((x 1 (1+ x)))
              (nil)
            (if (> x 5)
                (return-from nil x)
                (princ x)))))
12345
(A 6)
```

If `do` didn't create a block named `nil`, this example would have returned just `6`, rather than `(A 6)`.

The implicit block in `do` is not a problem, because `do` is advertised to behave this way. However, you should realize that if you write macros which expand into `do`s, they will capture the block name `nil`. In a macro like `for`, a `return` or `return-from nil` will return from the `for` expression, not the enclosing block.

9.9 Why Bother?

Some of the preceding examples are pretty pathological. Looking at them, one might be tempted to say "variable capture is so unlikely—why even worry about it?" There are two ways to answer this question. One is with another question: why write programs with small bugs when you could write programs with no bugs?

The longer answer is to point out that in real applications it's dangerous to assume anything about the way your code will be used. Any Lisp program has what is now called an "open architecture." If you're writing code other people will use, they may use it in ways you'd never anticipate. And it's not just people you have to worry about. Programs write programs too. It may be that no human would write code like

```
(before (progn (setq seq '(b a)) 'a)
        'b
        '(a b))
```

but code generated by programs often looks like this. Even if individual macros generate simple and reasonable-looking expansions, once you begin to nest macro calls, the expansions can become large programs which look like nothing any human would write. Under such circumstances, it is worth defending against cases, however contrived, which might make your macros expand incorrectly.

In the end, avoiding variable capture is not very difficult anyway. It soon becomes second-nature. The classic Common Lisp `defmacro` is like a cook's knife: an elegant idea which seems dangerous, but which experts use with confidence.

10

Other Macro Pitfalls

Writing macros requires an extra degree of caution. A function is isolated in its own lexical world, but a macro, because it is expanded into the calling code, can give the user an unpleasant surprise unless it is carefully written. Chapter 9 explained variable capture, the biggest such surprise. This chapter discusses four more problems to avoid when defining macros.

10.1 Number of Evaluations

Several incorrect versions of `for` appeared in the previous chapter. Figure 10.1 shows two more, accompanied by a correct version for comparison.

Though not vulnerable to capture, the second `for` contains a bug. It will generate an expansion in which the form passed as `stop` will be evaluated on each iteration. In the best case, this kind of macro is inefficient, repeatedly doing what it could have done just once. If `stop` has side-effects, the macro could actually produce incorrect results. For example, this loop will never terminate, because the goal recedes on each iteration:

```
> (let ((x 2))
    (for (i 1 (incf x))
      (princ i)))
12345678910111213...
```

In writing macros like `for`, one must remember that the arguments to a macro are forms, not values. Depending on where they appear in the expansion, they

A correct version:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

Subject to multiple evaluations:

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var)))
       ((> ,var ,stop))
     ,@body))
```

Incorrect order of evaluation:

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,gstop ,stop)
          (,var ,start (1+ ,var)))
         ((> ,var ,gstop))
       ,@body)))
```

Figure 10.1: Controlling argument evaluation.

could be evaluated more than once. In this case, the solution is to bind a variable to the value returned by the `stop` form, and refer to the variable during the loop.

Unless they are clearly intended for iteration, macros should ensure that expressions are evaluated exactly as many times as they appear in the macro call. There are obvious cases in which this rule does not apply: the Common Lisp `or` would be much less useful (it would become a Pascal `or`) if all its arguments were always evaluated. But in such cases the user knows how many evaluations to expect. This isn't so with the second version of `for`: the user has no reason to suppose that the `stop` form is evaluated more than once, and in fact there is no reason that it should be. A macro written like the second version of `for` is most likely written that way by mistake.

Unintended multiple evaluation is a particularly difficult problem for macros built on `setf`. Common Lisp provides several utilities to make writing such macros easier. The problem, and the solution, are discussed in Chapter 12.

10.2 Order of Evaluation

The order in which expressions are evaluated, though not as important as the number of times they are evaluated, can sometimes become an issue. In Common Lisp function calls, arguments are evaluated left-to-right:

```
> (setq x 10)
10
> (+ (setq x 3) x)
6
```

and it is good practice for macros to do the same. Macros should usually ensure that expressions are evaluated in the same order that they appear in the macro call.

In Figure 10.1, the third version of `for` also contains a subtle bug. The parameter `stop` will be evaluated before `start`, even though they appear in the opposite order in the macro call:

```
> (let ((x 1))
    (for (i x (setq x 13))
     (princ i)))
13
NIL
```

This macro gives a disconcerting impression of going back in time. The evaluation of the `stop` form influences the value returned by the `start` form, even though the `start` form appears first textually.

The correct version of `for` ensures that its arguments will be evaluated in the order in which they appear:

```
> (let ((x 1))
    (for (i x (setq x 13))
     (princ i)))
12345678910111213
NIL
```

Now setting `x` in the `stop` form has no effect on the value returned by the previous argument.

Although the preceding example is a contrived one, there are cases in which this sort of problem might really happen, and such a bug would be extremely difficult to find. Perhaps few people would write code in which the evaluation of one argument to a macro influenced the value returned by another, but people may do by accident things that they would never do on purpose. As well as having to work right when used as intended, a utility must not mask bugs. If anyone wrote code like the foregoing examples, it would probably be by mistake, but the correct version of `for` will make the mistake easier to detect.

10.3 Non-functional Expanders

Lisp expects code which generates macro expansions to be purely functional, in the sense described in Chapter 3. Expander code should depend on nothing but the forms passed to it as arguments, and should not try to have an effect on the world except by returning values.

As of CLTL2 (p. 685), it is safe to assume that macro calls in compiled code will not be re-expanded at runtime. Otherwise, Common Lisp makes no guarantees about when, or how often, a macro call will be expanded. It is considered an error for the expansion of a macro to vary depending on either. For example, suppose we wanted to count the number of times some macro is used. We can't simply do a search through the source files, because the macro might be called in code which is generated by the program. We might therefore want to define the macro as follows:

```
(defmacro nil! (x)                                      ; wrong
  (incf *nil!s*)
  `(setf ,x nil))
```

With this definition, the global `*nil!s*` will be incremented each time a call to `nil!` is expanded. However, we are mistaken if we expect the value of this variable to tell us how often `nil!` was called. A given call can be, and often is, expanded more than once. For example, a preprocessor which performed transformations on your source code might have to expand the macro calls in an expression before it could decide whether or not to transform it.

As a general rule, expander code shouldn't depend on anything except its arguments. So any macro which builds its expansion out of strings, for example, should be careful not to assume anything about what the package will be at the time of expansion. This concise but rather pathological example,

```
(defmacro string-call (opstring &rest args)             ; wrong
  `(,(intern opstring) ,@args))
```

defines a macro which takes the print name of an operator and expands into a call to it:

```
> (defun our+ (x y) (+ x y))
OUR+
> (string-call "OUR+" 2 3)
5
```

The call to `intern` takes a string and returns the corresponding symbol. However, if we omit the optional package argument, it does so in the current package. The expansion will thus depend on the package at the time the expansion is generated, and unless `our+` is visible in that package, the expansion will be a call to an unknown function.

Miller and Benson's *Lisp Style and Design* mentions one particularly ugly example of problems arising from side-effects in expander code. In Common Lisp, as of CLTL2 (p. 78), the lists bound to `&rest` parameters are not guaranteed to be freshly made. They may share structure with lists elsewhere in the program. In consequence, you shouldn't destructively modify `&rest` parameters, because you don't know what else you'll be modifying. ○

This possibility affects both functions and macros. With functions, problems would arise when using `apply`. In a valid implementation of Common Lisp the following could happen. Suppose we define a function `et-al`, which returns a list of its arguments with `et al` added to the end:

```
(defun et-al (&rest args)
  (nconc args (list 'et 'al)))
```

If we called this function normally, it would seem to work fine:

```
> (et-al 'smith 'jones)
(SMITH JONES ET AL)
```

However, if we called it via `apply`, it could alter existing data structures:

```
> (setq greats '(leonardo michelangelo))
(LEONARDO MICHELANGELO)
> (apply #'et-al greats)
(LEONARDO MICHELANGELO ET AL)
> greats
(LEONARDO MICHELANGELO ET AL)
```

At least, a valid implementation of Common Lisp could do this, though so far none seems to.

For macros, the danger is greater. A macro which altered an `&rest` parameter could thereby alter the macro call. That is, you could end up with inadvertently self-rewriting programs. The danger is also more real—it actually happens under existing implementations. If we define a macro which nconcs something onto its `&rest` argument[1]

```
(defmacro echo (&rest args)
  `',(nconc args (list 'amen)))
```

[1] `',(foo)` is equivalent to `` `(quote ,(foo)) ``.

and then define a function that calls it:

```
(defun foo () (echo x))
```

in one widely used Common Lisp, the following will happen:

```
> (foo)
(X AMEN AMEN)
> (foo)
(X AMEN AMEN AMEN)
```

Not only does `foo` return the wrong result, it returns a different result each time, because each macroexpansion alters the definition of `foo`.

This example also illustrates the point made earlier about multiple expansions of a given macro call. In this particular implementation, the first call to `foo` returns a lists with *two* `amen`s. For some reason this implementation expanded the macro call once when `foo` was defined, as well as once in each of the succeeding calls.

It would be safer to have defined `echo` as:

```
(defmacro echo (&rest args)
  `'(,@args amen))
```

because a comma-at is equivalent to an `append` rather than an `nconc`. After redefining this macro, `foo` will have to be redefined as well, even if it wasn't compiled, because the previous version of `echo` caused it to be rewritten.

In macros, it's not only `&rest` parameters which are subject to this danger. Any macro argument which is a list should be left alone. If we define a macro which modifies one of its arguments, and a function which calls it,

```
(defmacro crazy (expr) (nconc expr (list t)))

(defun foo () (crazy (list)))
```

then the source code of the calling function could get modified, as happens in one implementation the first time we call it:

```
> (foo)
(T T)
```

This happens in compiled as well as interpreted code.

The upshot is, don't try to avoid consing by destructively modifying parameter list structure. The resulting programs won't be portable, if they run at all. If you want to avoid consing in a function which takes a variable number of arguments, one solution is to use a macro, and thereby shift the consing forward to compile-time. For this application of macros, see Chapter 13.

One should also avoid performing destructive operations on the expressions returned by macro expanders, if these expressions incorporate quoted lists. This is not a restriction on macros *per se,* but an instance of the principle outlined in Section 3.3.

10.4 Recursion

Sometimes it's natural to define a function recursively. There's something inherently recursive about a function like this:

```
(defun our-length (x)
  (if (null x)
      0
      (1+ (our-length (cdr x)))))
```

This definition somehow seems more natural (though probably slower) than the iterative equivalent:

```
(defun our-length (x)
  (do ((len 0 (1+ len))
       (y x (cdr y)))
      ((null  y) len)))
```

A function which is neither recursive, nor part of some mutually recursive set of functions, can be transformed into a macro by the simple technique described in Section 7.10. However, just inserting backquotes and commas won't work with a recursive function. Let's take the built-in `nth` as an example. (For simplicity, our versions of `nth` will do no error-checking.) Figure 10.2 shows a mistaken attempt to define `nth` as a macro. Superficially, `nthb` appears to be equivalent to `ntha`, but a program containing a call to `nthb` would not compile, because the expansion of the call would never terminate.

In general, it's fine for macros to contain references to other macros, so long as expansion terminates somewhere. The trouble with `nthb` is that every expansion contains a reference to `nthb` itself. The function version, `ntha`, terminates because it recurses on the *value* of `n`, which is decremented on each recursion. But macroexpansion only has access to forms, not to their values. When the compiler tries to macroexpand, say, `(nthb x y)`, the first expansion will yield

```
(if (= x 0)
    (car y)
    (nthb (- x 1) (cdr y)))
```

This will work:

```
(defun ntha (n lst)
  (if (= n 0)
      (car lst)
      (ntha (- n 1) (cdr lst))))
```

This won't compile:

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
       (nthb (- ,n 1) (cdr ,lst))))
```

Figure 10.2: Mistaken analogy to a recursive function.

which will in turn expand into:

```
(if (= x 0)
    (car y)
    (if (= (- x 1) 0)
        (car (cdr y))
        (nthb (- (- x 1) 1) (cdr (cdr y)))))
```

and so on into an infinite loop. It's fine for a macro to expand into a call to itself, just so long as it doesn't always do so.

The dangerous thing about recursive macros like `nthb` is that they usually work fine under the interpreter. Then when you finally have your program working and you try to compile it, it won't even compile. Not only that, but there will usually be no indication that the problem is due to a recursive macro; the compiler will simply go into an infinite loop and leave you to figure out what went wrong.

In this case, `ntha` is tail-recursive. A tail-recursive function can easily be transformed into an iterative equivalent, and *then* used as a model for a macro. A macro like `nthb` could be written

```
(defmacro nthc (n lst)
  `(do ((n2 ,n (1- n2))
        (lst2 ,lst (cdr lst2)))
       ((= n2 0) (car lst2))))
```

so it is not impossible in principle to duplicate a recursive function with a macro. However, transforming more complicated recursive functions could be difficult, or even impossible.

```
(defmacro nthd (n lst)
  `(nth-fn ,n ,lst))

(defun nth-fn (n lst)
  (if (= n 0)
      (car lst)
      (nth-fn (- n 1) (cdr lst))))

(defmacro nthe (n lst)
  `(labels ((nth-fn (n lst)
              (if (= n 0)
                  (car lst)
                  (nth-fn (- n 1) (cdr lst)))))
     (nth-fn ,n ,lst)))
```

Figure 10.3: Two ways to fix the problem.

Depending on what you need a macro for, you may find it sufficient to use instead a combination of macro and function. Figure 10.3 shows two ways to make what appears to be a recursive macro. The first strategy, embodied by `nthd`, is simply to make the macro expand into a call to a recursive function. If, for example, you need a macro only to save users the trouble of quoting arguments, then this approach should suffice.

If you need a macro because you want its whole expansion to be inserted into the lexical environment of the macro call, then you would more likely want to follow the example of `nthe`. The built-in `labels` special form (Section 2.7) creates a local function definition. While each expansion of `nthc` will call the globally defined function `nth-fn`, each expansion of `nthe` will have its own version of such a function within it.

Although you can't translate a recursive function directly into a macro, you can write a macro whose expansion is recursively generated. The expansion function of a macro is a regular Lisp function, and can of course be recursive. For example, if we were to define a version of the built-in `or`, we would want to use a recursive expansion function.

Figure 10.4 shows two ways of defining recursive expansion functions for `or`. The macro `ora` calls the recursive function `or-expand` to generate its expansion. This macro will work, and so will the equivalent `orb`. Although `orb` recurses, it recurses on the arguments to the macro (which are available at macroexpansion time), not upon their values (which aren't). It might seem as if the expansion would contain a reference to `orb` itself, but the call to `orb` generated by one

```
(defmacro ora (&rest args)
  (or-expand args))

(defun or-expand (args)
  (if (null args)
       nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               ,(or-expand (cdr args)))))))

(defmacro orb (&rest args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               (orb ,@(cdr args)))))))
```

Figure 10.4: Recursive expansion functions.

macroexpansion step will be replaced by a `let` in the next one, yielding in the final expansion nothing more than a nested stack of `let`s; `(orb x y)` expands into code equivalent to:

```
(let ((g2 x))
  (if g2
      g2
      (let ((g3 y))
        (if g3 g3 nil))))
```

In fact, `ora` and `orb` are equivalent, and which style to use is just a matter of personal preference.

11

Classic Macros

This chapter shows how to define the most commonly used types of macros. They fall into three categories—with a fair amount of overlap. The first group are macros which create context. Any operator which causes its arguments to be evaluated in a new context will probably have to be defined as a macro. The first two sections describe the two basic types of context, and show how to define macros for each.

The next three sections describe macros for conditional and repeated evaluation. An operator whose arguments are to be evaluated less than once, or more than once, must also be defined as a macro. There is no sharp distinction between operators for conditional and repeated evaluation: some of the examples in this chapter do both (as well as binding). The final section explains another similarity between conditional and repeated evaluation: in some cases, both can be done with functions.

11.1 Creating Context

Context here has two senses. One sort of context is a lexical environment. The `let` special form creates a new lexical environment; the expressions in the body of a `let` will be evaluated in an environment which may contain new variables. If `x` is set to `a` at the toplevel, then

```
(let ((x 'b)) (list x))
```

will nonetheless return `(b)`, because the call to `list` will be made in an environment containing a new `x`, whose value is `b`.

```
(defmacro our-let (binds &body body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (consp x) (car x) x))
                     binds)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (consp x) (cadr x) nil))
              binds)))
```

Figure 11.1: Macro implementation of `let`.

An operator which is to have a body of expressions must usually be defined as a macro. Except for cases like `prog1` and `progn`, the purpose of such an operator will usually be to cause the body to be evaluated in some new context. A macro will be needed to wrap context-creating code around the body, even if the context does not include new lexical variables.

Figure 11.1 shows how `let` could be defined as a macro on `lambda`. An `our-let` expands into a function application—

```
(our-let ((x 1) (y 2))
  (+ x y))
```

expands into

```
((lambda (x y) (+ x y)) 1 2)
```

Figure 11.2 contains three new macros which establish lexical environments. Section 7.5 used `when-bind` as an example of parameter list destructuring, so this macro has already been described on page 94. The more general `when-bind*` takes a list of pairs of the form (*symbol expression*)—the same form as the first argument to `let`. If any *expression* returns `nil`, the whole `when-bind*` expression returns `nil`. Otherwise its body will be evaluated with each *symbol* bound as if by `let*`:

```
> (when-bind* ((x (find-if #'consp '(a (1 2) b)))
               (y (find-if #'oddp x)))
    (+ y 10))
11
```

Finally, the macro `with-gensyms` is itself for use in writing macros. Many macro definitions begin with the creation of gensyms, sometimes quite a number of them. The macro `with-redraw` (page 115) had to create five:

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))

(defmacro when-bind* (binds &body body)
  (if (null binds)
      `(progn ,@body)
      `(let (,(car binds))
         (if ,(caar binds)
             (when-bind* ,(cdr binds) ,@body)))))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))
```

Figure 11.2: Macros which bind variables.

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    ...))
```

Such definitions are simplified by `with-gensyms`, which binds a whole list of variables to gensyms. With the new macro we would write just:

```
(defmacro with-redraw ((var objs) &body body)
  (with-gensyms (gob x0 y0 x1 y1)
    ...))
```

This new macro will be used throughout the remaining chapters.

If we want to bind some variables and then, depending on some condition, evaluate one of a set of expressions, we just use a conditional within a `let`:

```
(let ((sun-place 'park) (rain-place 'library))
  (if (sunny)
      (visit sun-place)
      (visit rain-place)))
```

```
(defmacro condlet (clauses &body body)
  (let ((bodfn (gensym))
        (vars (mapcar #'(lambda (v) (cons v (gensym)))
                      (remove-duplicates
                        (mapcar #'car
                                (mappend #'cdr clauses))))))
    `(labels ((,bodfn ,(mapcar #'car vars)
                 ,@body))
       (cond ,@(mapcar #'(lambda (cl)
                           (condlet-clause vars cl bodfn))
                       clauses)))))

(defun condlet-clause (vars cl bodfn)
  `(,(car cl) (let ,(mapcar #'cdr vars)
                (let ,(condlet-binds vars cl)
                  (,bodfn ,@(mapcar #'cdr vars))))))


(defun condlet-binds (vars cl)
  (mapcar #'(lambda (bindform)
              (if (consp bindform)
                  (cons (cdr (assoc (car bindform) vars))
                        (cdr bindform))))
          (cdr cl)))
```

Figure 11.3: Combination of `cond` and `let`.

Unfortunately, there is no convenient idiom for the opposite situation, where we always want to evaluate the same code, but where the *bindings* must vary depending on some condition.

Figure 11.3 contains a macro intended for such situations. As its name suggests, `condlet` behaves like the offspring of `cond` and `let`. It takes as arguments a list of binding clauses, followed by a body of code. Each of the binding clauses is guarded by a test expression; the body of code will be evaluated with the bindings specified by the first binding clause whose test expression returns true. Variables which occur in some clauses and not others will be bound to `nil` if the successful clause does not specify bindings for them:

```
> (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
           ((= 1 1) (y (princ 'c)) (x (princ 'd)))
           (t       (x (princ 'e)) (z (princ 'f))))
    (list x y z))
CD
(D C NIL)
```

The definition of `condlet` can be understood as a generalization of the definition of `our-let`. The latter makes its body into a function, which is applied to the results of evaluating the initial-value forms. A `condlet` expands into code which defines a local function with `labels`; within it a `cond` clause determines which set of initial-value forms will be evaluated and passed to the function.

Notice that the expander uses `mappend` instead of `mapcan` to extract the variable names from the binding clauses. This is because `mapcan` is destructive, and as Section 10.3 warned, it is dangerous to modify parameter list structure.

11.2 The `with-` Macro

There is another kind of context besides a lexical environment. In the broader sense, the context is the state of the world, including the values of special variables, the contents of data structures, and the state of things outside Lisp. Operators which build this kind of context must be defined as macros too, unless their code bodies are to be packaged up in closures.

The names of context-building macros often begin with `with-`. The most commonly used macro of this type is probably `with-open-file`. Its body is evaluated with a newly opened file bound to a user-supplied variable:

```
(with-open-file (s "dump" :direction :output)
  (princ 99 s))
```

After evaluation of this expression the file `"dump"` will automatically be closed, and its contents will be the two characters `"99"`.

This operator clearly has to be defined as a macro, because it binds `s`. However, operators which cause forms to be evaluated in a new context must be defined as macros anyway. The `ignore-errors` macro, new in CLTL2, causes its arguments to be evaluated as if in a `progn`. If an error occurs at any point, the whole `ignore-errors` form simply returns `nil`. (This would be useful, for example, when reading input typed by the user.) Though `ignore-errors` creates no variables, it still must be defined as a macro, because its arguments are evaluated in a new context.

Generally, macros which create context will expand into a block of code; additional expressions may be placed before the body, after it, or both. If code occurs after the body, its purpose may be to leave the system in a consistent state—to clean up something. For example, `with-open-file` has to close the file it opened. In such situations, it is typical to make the context-creating macro expand into an `unwind-protect`.

The purpose of `unwind-protect` is to ensure that certain expressions are evaluated even if execution is interrupted. It takes one or more arguments, which are evaluated in order. If all goes smoothly it will return the value of the first argument, like a `prog1`. The difference is, the remaining arguments will be evaluated even if an error or throw interrupts evaluation of the first.

```
> (setq x 'a)
A
> (unwind-protect
    (progn (princ "What error?")
           (error "This error."))
    (setq x 'b))
What error?
>>Error: This error.
```

The `unwind-protect` form as a whole yields an error. However, after returning to the toplevel, we notice that the second argument still got evaluated:

```
> x
B
```

Because `with-open-file` expands into an `unwind-protect`, the file it opens will usually be closed even if an error occurs during the evaluation of its body.

Context-creating macros are mostly written for specific applications. As an example, suppose we are writing a program which deals with multiple, remote databases. The program talks to one database at a time, indicated by the global variable `*db*`. Before using a database, we have to lock it, so that no one else can use it at the same time. When we are finished we have to release the lock. If we want the value of the query `q` on the database `db`, we might say something like:

```
(let ((temp *db*))
  (setq *db* db)
  (lock *db*)
  (prog1 (eval-query q)
         (release *db*)
         (setq *db* temp)))
```

With a macro we can hide all this bookkeeping. Figure 11.4 defines a macro which will allow us to deal with databases at a higher level of abstraction. Using `with-db`, we would say just:

Pure macro:

```
(defmacro with-db (db &body body)
  (let ((temp (gensym)))
    `(let ((,temp *db*))
       (unwind-protect
         (progn
           (setq *db* ,db)
           (lock *db*)
           ,@body)
         (progn
           (release *db*)
           (setq *db* ,temp))))))
```

Combination of macro and function:

```
(defmacro with-db (db &body body)
  (let ((gbod (gensym)))
    `(let ((,gbod #'(lambda () ,@body)))
       (declare (dynamic-extent ,gbod))
       (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
  (unwind-protect
    (progn
      (setq *db* new-db)
      (lock *db*)
      (funcall body))
    (progn
      (release *db*)
      (setq *db* old-db))))
```

Figure 11.4: A typical `with-` macro.

```
(with-db db
  (eval-query q))
```

Calling `with-db` is also safer, because it expands into an `unwind-protect` instead of a simple `prog1`.

The two definitions of `with-db` in Figure 11.4 illustrate two possible ways to write this kind of macro. The first is a pure macro, the second a combination of a function and a macro. The second approach becomes more practical as the

```
(defmacro if3 (test t-case nil-case ?-case)
  `(case ,test
     ((nil) ,nil-case)
     (?     ,?-case)
     (t     ,t-case)))

(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
             ((zerop ,g) ,zero)
             (t ,neg)))))
```

Figure 11.5: Macros for conditional evaluation.

desired `with-` macro grows in complexity.

In CLTL2 Common Lisp, the `dynamic-extent` declaration allows the closure containing the body to be allocated more efficiently (in CLTL1 implementations, it will be ignored). We only need this closure for the duration of the call to `with-db-fn`, and the declaration says as much, allowing the compiler to allocate space for it on the stack. This space will be reclaimed automatically on exit from the `let` expression, instead of being reclaimed later by the garbage-collector.

11.3 Conditional Evaluation

Sometimes we want an argument in a macro call to be evaluated only under certain conditions. This is beyond the ability of functions, which always evaluate all their arguments. Built-in operators like `if`, `and`, and `cond` protect some of their arguments from evaluation unless other arguments return certain values. For example, in this expression

```
(if t
    'phew
    (/ x 0))
```

the third argument would cause a division-by-zero error if it were evaluated. But since only the first two arguments ever will be evaluated, the `if` as a whole will always safely return `phew`.

We can create new operators of this sort by writing macros which expand into calls to the existing ones. The two macros in Figure 11.5 are two of many possible variations on `if`. The definition of `if3` shows how we could define a conditional for a three-valued logic. Instead of treating `nil` as false and everything else as true, this macro considers three categories of truth: true, false, and *uncertain*, represented as `?`. It might be used as in the following description of a five year-old:

```
(while (not sick)
  (if3 (cake-permitted)
       (eat-cake)
       (throw 'tantrum nil)
       (plead-insistently)))
```

The new conditional expands into a `case`. (The `nil` key has to be enclosed within a list because a `nil` key alone would be ambiguous.) Only one of the last three arguments will be evaluated, depending on the value of the first.

The name `nif` stands for "numeric if." Another implementation of this macro appeared on page 86. It takes a numeric expression as its first argument, and depending on its sign evaluates one of the remaining three arguments.

```
> (mapcar #'(lambda (x)
              (nif x 'p 'z 'n))
          '(0 1 -1))
(Z P N)
```

Figure 11.6 contains several more macros which take advantage of conditional evaluation. The macro `in` is to test efficiently for set membership. When you want to test whether an object is one of a set of alternatives, you could express the query as a disjunction:

```
(let ((x (foo)))
  (or (eql x (bar)) (eql x (baz))))
```

or you could express it in terms of set membership:

```
(member (foo) (list (bar) (baz)))
```

The latter is more abstract, but less efficient. The `member` expression incurs unnecessary costs from two sources. It conses, because it must assemble the alternatives into a list for `member` to search. And to form the alternatives into a list they all have to be evaluated, even though some of the values may never be needed. If the value of `(foo)` is equal to the value of `(bar)`, then there is no need to evaluate `(baz)`. Whatever its conceptual advantages, this is not a good way to use `member`. We can get the same abstraction more efficiently with a macro: `in` combines the abstraction of `member` with the efficiency of `or`. The equivalent `in` expression

```
(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro inq (obj &rest args)
  `(in ,obj ,@(mapcar #'(lambda (a)
                          `',a)
                      args)))

(defmacro in-if (fn &rest choices)
  (let ((fnsym (gensym)))
    `(let ((,fnsym ,fn))
       (or ,@(mapcar #'(lambda (c)
                         `(funcall ,fnsym ,c))
                     choices)))))

(defmacro >case (expr &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ,@(mapcar #'(lambda (cl) (>casex g cl))
                       clauses)))))

(defun >casex (g cl)
  (let ((key (car cl)) (rest (cdr cl)))
    (cond ((consp key) `((in ,g ,@key) ,@rest))
          ((inq key t otherwise) `(t ,@rest))
          (t (error "bad >case clause")))))
```

Figure 11.6: Macros for conditional evaluation.

```
(in (foo) (bar) (baz))
```

has the same shape as the `member` expression, but expands into

```
(let ((#:g25 (foo)))
  (or (eql #:g25 (bar))
      (eql #:g25 (baz))))
```

As is often the case, when faced with a choice between a clean idiom and an efficient one, we go between the horns of the dilemma by writing a macro which transforms the former into the latter.

Pronounced “in queue,” `inq` is a quoting variant of `in`, as `setq` used to be of `set`. The expression

```
(inq operator + - *)
```

expands into

```
(in operator '+ '- '*)
```

As `member` does by default, `in` and `inq` use `eql` to test for equality. When you want to use some other test—or any other function of one argument—you can use the more general `in-if`. What `in` is to `member`, `in-if` is to `some`. The expression

```
(member x (list a b) :test #'equal)
```

can be duplicated by

```
(in-if #'(lambda (y) (equal x y)) a b)
```

and

```
(some #'oddp (list a b))
```

becomes

```
(in-if #'oddp a b)
```

Using a combination of `cond` and `in`, we can define a useful variant of `case`. The Common Lisp `case` macro assumes that its keys are constants. Sometimes we may want the behavior of a `case` expression, but with keys which are evaluated. For such situations we define `>case`, like `case` except that the keys guarding each clause are evaluated before comparison. (The `>` in the name is intended to suggest the arrow notation used to represent evaluation.) Because `>case` uses `in`, it evaluates no more of the keys than it needs to.

Since keys can be Lisp expressions, there is no way to tell if `(x y)` is a call or a list of two keys. To avoid ambiguity, keys (other than `t` and `otherwise`) must always be given in a list, even if there is only one of them. In `case` expressions, `nil` may not appear as the car of a clause on grounds of ambiguity. In a `>case` expression, `nil` is no longer ambiguous as the car of a clause, but it does mean that the rest of the clause will never be evaluated.

For clarity, the code that generates the expansion of each `>case` clause is defined as a separate function, `>casex`. Notice that `>casex` itself uses `inq`.

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))

(defmacro till (test &body body)
  `(do ()
       (,test)
     ,@body))

(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

Figure 11.7: Simple iteration macros.

11.4 Iteration

Sometimes the trouble with functions is not that their arguments are always evaluated, but that they are evaluated only once. Because each argument to a function will be evaluated exactly once, if we want to define an operator which takes some body of expressions and iterates through them, we will have to define it as a macro.

The simplest example would be a macro which evaluated its arguments in sequence forever:

```
(defmacro forever (&body body)
  `(do ()
       (nil)
     ,@body))
```

This is just what the built-in `loop` macro does if you give it no loop keywords. It might seem that there is not much future (or too much future) in looping forever. But combined with `block` and `return-from`, this kind of macro becomes the most natural way to express loops where termination is always in the nature of an emergency.

Some of the simplest macros for iteration are shown in Figure 11.7. We have already seen `while` (page 91), whose body will be evaluated while a test expression returns true. Its converse is `till`, which does the same while a test expression returns false. Finally `for`, also seen before (page 129), iterates for a range of numbers.

By defining these macros to expand into `do`s, we enable the use of `go` and `return` within their bodies. As `do` inherits these rights from `block` and `tagbody`, `while`, `till`, and `for` inherit them from `do`. As explained on page 131, the `nil` tag of the implicit `block` around `do` will be captured by the macros defined in Figure 11.7. This is more of a feature than a bug, but it should at least be mentioned explicitly.

Macros are indispensable when we need to define more powerful iteration constructs. Figure 11.8 contains two generalizations of `dolist`; both evaluate their body with a tuple of variables bound to successive subsequences of a list. For example, given two parameters, `do-tuples/o` will iterate by pairs:

```
> (do-tuples/o (x y) '(a b c d)
    (princ (list x y)))
(A B)(B C)(C D)
NIL
```

Given the same arguments, `do-tuples/c` will do the same thing, then wrap around to the front of the list:

```
> (do-tuples/c (x y) '(a b c d)
    (princ (list x y)))
(A B)(B C)(C D)(D A)
NIL
```

Both macros return `nil`, unless an explicit `return` occurs within the body.

This kind of iteration is often needed in programs which deal with some notion of a path. The suffixes `/o` and `/c` are intended to suggest that the two versions traverse open and closed paths, respectively. For example, if `points` is a list of points and `(drawline` *x y*`)` draws the line between *x* and *y,* then to draw the path from the first point to the last we write.

```
(do-tuples/o (x y) points (drawline x y))
```

whereas, if `points` is a list of the vertices of a polygon, to draw its perimeter we write

```
(do-tuples/c (x y) points (drawline x y))
```

The list of parameters given as the first argument can be any length, and iteration will proceed by tuples of that length. If just one parameter is given, both

```
(defmacro do-tuples/o (parms source &body body)
  (if parms
      (let ((src (gensym)))
        `(prog ((,src ,source))
           (mapc #'(lambda ,parms ,@body)
                 ,@(map0-n #'(lambda (n)
                               `(nthcdr ,n ,src))
                           (1- (length parms))))))))

(defmacro do-tuples/c (parms source &body body)
  (if parms
      (with-gensyms (src rest bodfn)
        (let ((len (length parms)))
          `(let ((,src ,source))
             (when (nthcdr ,(1- len) ,src)
               (labels ((,bodfn ,parms ,@body))
                 (do ((,rest ,src (cdr ,rest)))
                     ((not (nthcdr ,(1- len) ,rest))
                      ,@(mapcar #'(lambda (args)
                                    `(,bodfn ,@args))
                                (dt-args len rest src))
                      nil)
                   (,bodfn ,@(map1-n #'(lambda (n)
                                         `(nth ,(1- n)
                                               ,rest))
                                     len))))))))))

(defun dt-args (len rest src)
  (map0-n #'(lambda (m)
              (map1-n #'(lambda (n)
                          (let ((x (+ m n)))
                            (if (>= x len)
                                `(nth ,(- x len) ,src)
                                `(nth ,(1- x) ,rest))))
                      len))
          (- len 2)))
```

Figure 11.8: Macros for iteration by subsequences.

```
(do-tuples/c (x y z) '(a b c d)
  (princ (list x y z)))
```

expands into:

```
(let ((#:g2 '(a b c d)))
  (when (nthcdr 2 #:g2)
    (labels ((#:g4 (x y z)
               (princ (list x y z))))
      (do ((#:g3 #:g2 (cdr #:g3)))
          ((not (nthcdr 2 #:g3))
           (#:g4 (nth 0 #:g3)
                 (nth 1 #:g3)
                 (nth 0 #:g2))
           (#:g4 (nth 1 #:g3)
                 (nth 0 #:g2)
                 (nth 1 #:g2))
           nil)
        (#:g4 (nth 0 #:g3)
              (nth 1 #:g3)
              (nth 2 #:g3))))))
```

Figure 11.9: Expansion of a call to `do-tuples/c`.

degenerate to `dolist`:

```
> (do-tuples/o (x) '(a b c) (princ x))
ABC
NIL
> (do-tuples/c (x) '(a b c) (princ x))
ABC
NIL
```

The definition of `do-tuples/c` is more complex than that of `do-tuples/o`, because it has to wrap around on reaching the end of the list. If there are n parameters, `do-tuples/c` must do $n-1$ more iterations before returning:

```
> (do-tuples/c (x y z) '(a b c d)
    (princ (list x y z)))
(A B C)(B C D)(C D A)(D A B)
NIL
```

```
> (do-tuples/c (w x y z) '(a b c d)
    (princ (list w x y z)))
(A B C D)(B C D A)(C D A B)(D A B C)
NIL
```

The expansion of the former call to `do-tuples/c` is shown in Figure 11.9. The hard part to generate is the sequence of calls representing the wrap around to the front of the list. These calls (in this case, two of them) are generated by `dt-args`.

11.5 Iteration with Multiple Values

The built-in `do` macros have been around longer than multiple return values. Fortunately `do` can evolve to suit the new situation, because the evolution of Lisp is in the hands of the programmer. Figure 11.10 contains a version of `do*` adapted for multiple values. With `mvdo*`, each of the initial clauses can bind more than one variable:

```
> (mvdo* ((x 1 (1+ x))
          ((y z) (values 0 0) (values z x)))
         ((> x 5) (list x y z))
    (princ (list x y z)))
(1 0 0)(2 0 2)(3 2 3)(4 3 4)(5 4 5)
(6 5 6)
```

This kind of iteration is useful, for example, in interactive graphics programs, which often have to deal with multiple quantities like coordinates and regions.

Suppose that we want to write a simple interactive game, in which the object is to avoid being squashed between two pursuing objects. If the two pursuers both hit you at the same time, you lose; if they crash into one another first, you win. Figure 11.11 shows how the main loop of this game could be written using `mvdo*`.

It is also possible to write an `mvdo`, which binds its local variables in parallel:

```
> (mvdo ((x 1 (1+ x))
         ((y z) (values 0 0) (values z x)))
        ((> x 5) (list x y z))
    (princ (list x y z)))
(1 0 0)(2 0 1)(3 1 2)(4 2 3)(5 3 4)
(6 4 5)
```

The need for `psetq` in defining `do` was described on page 96. To define `mvdo`, ◦ we need a multiple-value version of `psetq`. Since Common Lisp doesn't have one, we have to write it ourselves, as in Figure 11.12. The new macro works as follows:

```
(defmacro mvdo* (parm-cl test-cl &body body)
  (mvdo-gen parm-cl parm-cl test-cl body))

(defun mvdo-gen (binds rebinds test body)
  (if (null binds)
      (let ((label (gensym)))
        `(prog nil
           ,label
           (if ,(car test)
               (return (progn ,@(cdr test))))
           ,@body
           ,@(mvdo-rebind-gen rebinds)
           (go ,label)))
      (let ((rec (mvdo-gen (cdr binds) rebinds test body)))
        (let ((var/s (caar binds)) (expr (cadar binds)))
          (if (atom var/s)
              `(let ((,var/s ,expr)) ,rec)
              `(multiple-value-bind ,var/s ,expr ,rec))))))

(defun mvdo-rebind-gen (rebinds)
  (cond ((null rebinds) nil)
        ((< (length (car rebinds)) 3)
         (mvdo-rebind-gen (cdr rebinds)))
        (t
         (cons (list (if (atom (caar rebinds))
                         'setq
                         'multiple-value-setq)
                     (caar rebinds)
                     (third (car rebinds)))
               (mvdo-rebind-gen (cdr rebinds))))))
```

Figure 11.10: Multiple value binding version of do*.

```
> (let ((w 0) (x 1) (y 2) (z 3))
    (mvpsetq (w x) (values 'a 'b) (y z) (values w x))
    (list w x y z))
(A B 0 1)
```

The definition of mvpsetq relies on three utility functions: mklist (page 45), group (page 47), and shuffle, defined here, which interleaves two lists:

```
(mvdo* (((px py) (pos player)   (move player mx my))
        ((x1 y1) (pos obj1)     (move obj1 (- px x1)
                                           (- py y1)))
        ((x2 y2) (pos obj2)     (move obj2 (- px x2)
                                           (- py y2)))
        ((mx my) (mouse-vector) (mouse-vector))
        (win     nil            (touch obj1 obj2))
        (lose    nil            (and (touch obj1 player)
                                     (touch obj2 player))))
       ((or win lose) (if win 'win 'lose))
  (clear)
  (draw obj1)
  (draw obj2)
  (draw player))
```

(`pos` *obj*) returns two values `x,y` representing the position of *obj*. Initially, the three objects have random positions.

(`move` *obj dx dy*) moves the object *obj* depending on its type and the vector ⟨*dx,dy*⟩. Returns two values `x,y` indicating the new position.

(`mouse-vector`) returns two values `dx,dy` indicating the current movement of the mouse.

(`touch` *obj1 obj2*) returns true if *obj1* and *obj2* are touching.

(`clear`) clears the game region.

(`draw` *obj*) draws *obj* at its current position.

Figure 11.11: A game of squash.

```
> (shuffle '(a b c) '(1 2 3 4))
(A 1 B 2 C 3 4)
```

With `mvpsetq`, we can define `mvdo` as in Figure 11.13. Like `condlet`, this macro uses `mappend` instead of `mapcar` to avoid modifying the original macro call. The `mappend-mklist` idiom flattens a tree by one level:

```
> (mappend #'mklist '((a b c) d (e (f g) h) ((i)) j))
(A B C D E (F G) H (I) J)
```

```
(defmacro mvpsetq (&rest args)
  (let* ((pairs (group args 2))
         (syms  (mapcar #'(lambda (p)
                            (mapcar #'(lambda (x) (gensym))
                                    (mklist (car p))))
                        pairs)))
    (labels ((rec (ps ss)
               (if (null ps)
                   `(setq
                     ,@(mapcan #'(lambda (p s)
                                   (shuffle (mklist (car p))
                                            s))
                               pairs syms))
                   (let ((body (rec (cdr ps) (cdr ss))))
                     (let ((var/s (caar ps))
                           (expr (cadar ps)))
                       (if (consp var/s)
                           `(multiple-value-bind ,(car ss)
                                                 ,expr
                              ,body)
                           `(let ((,@(car ss) ,expr))
                              ,body)))))))
      (rec pairs syms))))

(defun shuffle (x y)
  (cond ((null x) y)
        ((null y) x)
        (t (list* (car x) (car y)
                  (shuffle (cdr x) (cdr y))))))
```

Figure 11.12: Multiple value version of psetq.

To help in understanding this rather large macro, Figure 11.14 contains a sample expansion.

11.6 Need for Macros

Macros aren't the only way to protect arguments against evaluation. Another is to wrap them in closures. Conditional and repeated evaluation are similar because neither problem inherently requires macros. For example, we could write a version

```
(defmacro mvdo (binds (test &rest result) &body body)
  (let ((label (gensym))
        (temps (mapcar #'(lambda (b)
                           (if (listp (car b))
                               (mapcar #'(lambda (x)
                                           (gensym))
                                       (car b))
                               (gensym)))
                       binds)))
    `(let ,(mappend #'mklist temps)
       (mvpsetq ,@(mapcan #'(lambda (b var)
                              (list var (cadr b)))
                          binds
                          temps))
       (prog ,(mapcar #'(lambda (b var) (list b var))
                      (mappend #'mklist (mapcar #'car binds))
                      (mappend #'mklist temps))
         ,label
         (if ,test
             (return (progn ,@result)))
         ,@body
         (mvpsetq ,@(mapcan #'(lambda (b)
                                (if (third b)
                                    (list (car b)
                                          (third b))))
                            binds))
         (go ,label)))))
```

Figure 11.13: Multiple value binding version of do.

of `if` as a function:

```
(defun fnif (test then &optional else)
  (if test
      (funcall then)
      (if else (funcall else))))
```

We would protect the `then` and `else` arguments by expressing them as closures, so instead of

```
(if (rich) (go-sailing) (rob-bank))
```

```
(mvdo ((x 1 (1+ x))
       ((y z) (values 0 0) (values z x)))
      ((> x 5) (list x y z))
  (princ (list x y z)))
```

expands into:

```
(let (#:g2 #:g3 #:g4)
  (mvpsetq #:g2 1
           (#:g3 #:g4) (values 0 0))
  (prog ((x #:g2) (y #:g3) (z #:g4))
     #:g1
     (if (> x 5)
         (return (progn (list x y z))))
     (princ (list x y z))
     (mvpsetq x (1+ x)
              (y z) (values z x))
     (go #:g1)))
```

Figure 11.14: Expansion of a call to `mvdo`.

we would say

```
(fnif (rich)
      #'(lambda () (go-sailing))
      #'(lambda () (rob-bank)))
```

If all we want is conditional evaluation, macros aren't absolutely necessary. They just make programs cleaner. However, macros are necessary when we want to take apart argument forms, or bind variables passed as arguments.

The same applies to macros for iteration. Although macros offer the only way to define an iteration construct which can be followed by a body of expressions, it is possible to do iteration with functions, so long as the body of the loop is packaged up in a function itself.[1] The built-in function `mapc`, for example, is the functional counterpart of `dolist`. The expression

```
(dolist (b bananas)
  (peel b)
  (eat b))
```

[1]It's not *impossible* to write an iteration function which doesn't need its argument wrapped up in a function. We could write a function that called `eval` on expressions passed to it as arguments. For an explanation of why it's usually bad to call `eval`, see page 278.

has the same side-effects as

```
(mapc #'(lambda (b)
          (peel b)
          (eat b))
      bananas)
```

(though the former returns `nil` and the latter returns the list `bananas`). We could likewise implement `forever` as a function,

```
(defun forever (fn)
  (do ()
      (nil)
    (funcall fn)))
```

if we were willing to pass it a closure instead of a body of expressions.

However, iteration constructs usually want to do more than just iterate, as `forever` does: they usually want to do a combination of binding and iteration. With a function, the prospects for binding are limited. If you want to bind variables to successive elements of lists, you can use one of the mapping functions. But if the requirements get much more complicated than that, you'll have to write a macro.

12

Generalized Variables

Chapter 8 mentioned that one of the advantages of macros is their ability to transform their arguments. One macro of this sort is `setf`. This chapter looks at the implications of `setf`, and then shows some examples of macros which can be built upon it.

Writing correct macros on `setf` is surprisingly difficult. To introduce the topic, the first section will provide a simple example which is slightly incorrect. The next section will explain what's wrong with this macro, and show how to fix it. The third and fourth sections present examples of utilities built on `setf`, and the final section explains how to define your own `setf` inversions.

12.1 The Concept

The built-in macro `setf` is a generalization of `setq`. The first argument to `setf` can be a call instead of just a variable:

```
> (setq lst '(a b c))
(A B C)
> (setf (car lst) 480)
480
> lst
(480 B C)
```

In general `(setf` *x* *y*`)` can be understood as saying "see to it that *x* evaluates to *y*." As a macro, `setf` can look inside its arguments to see what needs to be done to make such a statement true. If the first argument (after macroexpansion) is a symbol, the `setf` just expands into a `setq`. But if the first argument is a query, the `setf` expands into the corresponding assertion. Since the second argument is a constant, the preceding example could expand into:

```
(progn (rplaca lst 480) 480)
```

This transformation from query to assertion is called *inversion.* All the most frequently used Common Lisp access functions have predefined inversions, including `car`, `cdr`, `nth`, `aref`, `get`, `gethash`, and the access functions created by `defstruct`. (The full list is in CLTL2, p. 125.)

An expression which can serve as the first argument to `setf` is called a *generalized variable.* Generalized variables have turned out to be a powerful abstraction. A macro call resembles a generalized variable in that any macro call which expands into an invertible reference will itself be invertible.

When we also write our own macros on top of `setf`, the combination leads to noticeably cleaner programs. One of the macros we can define on top of `setf` is `toggle`,[1]

```
(defmacro toggle (obj)                              ; wrong
  `(setf ,obj (not ,obj)))
```

which toggles the value of a generalized variable:

```
> (let ((lst '(a b c)))
    (toggle (car lst))
    lst)
(NIL B C)
```

Now consider the following sample application. Suppose someone—a soap-opera writer, energetic busybody, or party official—wants to maintain a database of all the relations between the inhabitants of a small town. Among the tables required is one which records people's friends:

```
(defvar *friends* (make-hash-table))
```

The entries in this hash-table are themselves hash-tables, in which names of potential friends are mapped to `t` or `nil`:

```
(setf (gethash 'mary *friends*) (make-hash-table))
```

To make John the friend of Mary, we would say:

```
(setf (gethash 'john (gethash 'mary *friends*)) t)
```

[1]This definition is not correct, as the following section will explain.

The town is divided between two factions. As factions are wont to do, each says "anyone who is not with us is against us," so everyone in town has been compelled to join one side or the other. Thus when someone switches sides, all his friends become enemies and all his enemies become friends.

To toggle whether x is the friend of y using only built-in operators, we have to say:

```
(setf (gethash x (gethash y *friends*))
      (not (gethash x (gethash y *friends*))))
```

which is a rather complicated expression, though much simpler than it would have been without setf. If we had defined an access macro upon the database as follows:

```
(defmacro friend-of (p q)
  `(gethash ,p (gethash ,q *friends*)))
```

then between this macro and toggle, we would have been better equipped to deal with changes to the database. The previous update could have been expressed as simply:

```
(toggle (friend-of x y))
```

Generalized variables are like a health food that tastes good. They yield programs which are virtuously modular, and yet beautifully elegant. If you provide access to your data structures through macros or invertible functions, other modules can use setf to modify your data structures without having to know the details of their representation.

12.2 The Multiple Evaluation Problem

The previous section warned that our initial definition of toggle was incorrect:

```
(defmacro toggle (obj)                                    ; wrong
  `(setf ,obj (not ,obj)))
```

It is subject to the problem described in Section 10.1, multiple evaluation. Trouble arises when its argument has side-effects. For example, if lst is a list of objects, and we write:

```
(toggle (nth (incf i) lst))
```

then we would expect to be toggling the (i+1)th element. However, with the current definition of toggle this call will expand into:

```
(setf (nth (incf i) lst)
      (not (nth (incf i) lst)))
```

This increments `i` twice, and sets the (i+1)th element to the opposite of the (i+2)th element. So in this example

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(T NIL T)
```

the call to `toggle` seems to have no effect.

It is not enough just to take the expression given as an argument to `toggle` and insert it as the first argument to `setf`. We have to look inside the expression to see what it does: if it contains subforms, we have to break them apart and evaluate them separately, in case they have side effects. In general, this is a complicated business.

To make it easier, Common Lisp provides a macro which automatically defines a limited class of macros on `setf`. This macro is called `define-modify-macro`, and it takes three arguments: the name of the macro, its additional parameters (after the generalized variable), and the name of the function[2] which yields the new value for the generalized variable.

Using `define-modify-macro`, we could define `toggle` as follows:

```
(define-modify-macro toggle () not)
```

Paraphrased, this says "to evaluate an expression of the form `(toggle` *place*`)`, find the location specified by *place*, and if the value stored there is *val*, replace it with the value of `(not` *val*`)`." Here is the new macro used in the same example:

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(NIL NIL T)
```

This version gives the correct result, but it could be made more general. Since `setf` and `setq` can take an arbitrary number of arguments, so should `toggle`. We can add this capability by defining another macro on top of the modify-macro, as in Figure 12.1.

[2] A function name in the general sense: either `1+` or `(lambda (x) (+ x 1))`.

```
(defmacro allf (val &rest args)
  (with-gensyms (gval)
    `(let ((,gval ,val))
       (setf ,@(mapcan #'(lambda (a) (list a gval))
                        args)))))

(defmacro nilf (&rest args) `(allf nil ,@args))

(defmacro tf (&rest args) `(allf t ,@args))

(defmacro toggle (&rest args)
  `(progn
     ,@(mapcar #'(lambda (a) `(toggle2 ,a))
               args)))

(define-modify-macro toggle2 () not)
```

Figure 12.1: Macros which operate on generalized variables.

12.3 New Utilities

This section gives some examples of new utilities which operate on generalized variables. They must be macros in order to pass their arguments intact to `setf`.

Figure 12.1 shows four new macros built upon `setf`. The first, `allf`, is for setting a number of generalized variables to the same value. Upon it are built `nilf` and `tf`, which set their arguments to `nil` and `t`, respectively. These macros are simple, but they make a difference.

Like `setq`, `setf` can take multiple arguments—alternating variables and values:

```
(setf x 1 y 2)
```

So can these new utilities, but you can skip giving half the arguments. If you want to initialize a number of variables to `nil`, instead of

```
(setf x nil y nil z nil)
```

you can say just

```
(nilf x y z)
```

```
(define-modify-macro concf (obj) nconc)

(define-modify-macro conc1f (obj)
  (lambda (place obj)
    (nconc place (list obj))))

(define-modify-macro concnew (obj &rest args)
  (lambda (place obj &rest args)
    (unless (apply #'member obj place args)
      (nconc place (list obj)))))
```

Figure 12.2: List operations on generalized variables.

The last macro, `toggle`, was described in the previous section: it is like `nilf`, but gives each of its arguments the opposite truth value.

These four macros illustrate an important point about operators for assignment. Even if we only intend to use an operator on ordinary variables, it's worth writing it to expand into a `setf` instead of a `setq`. If the first argument is a symbol, the `setf` will expand into a `setq` anyway. Since we can have the generality of `setf` at no extra cost, it is rarely desirable to use `setq` in a macroexpansion.

Figure 12.2 contains three macros for destructively modifying the ends of lists. Section 3.1 mentioned that it is unsafe to rely on

```
(nconc x y)
```

for side-effects, and that one must write instead

```
(setq x (nconc x y))
```

This idiom is embodied in `concf`. The more specialized `conc1f` and `concnew` are like `push` and `pushnew` for the other end of the list: `conc1f` adds one element to the end of a list, and `concnew` does the same, but only if the element is not already a member.

Section 2.2 mentioned that the name of a function can be a lambda-expression as well as a symbol. Thus it is fine to give a whole lambda-expression as the third argument to `define-modify-macro`, as in the definition of `conc1f`. Using `conc1` from page 45, this macro could also have been written:

```
(define-modify-macro conc1f (obj) conc1)
```

The macros in Figure 12.2 should be used with one reservation. If you're planning to build a list by adding elements to the end, it may be preferable to use push, and then nreverse the list. It is cheaper to do something to the front of a list than to the end, because to do something to the end you have to get there first. It is probably to encourage efficient programming that Common Lisp has many operators for the former and few for the latter.

12.4 More Complex Utilities

Not all macros on setf can be defined with define-modify-macro. Suppose, for example, that we want to define a macro _f for applying a function destructively to a generalized variable. The built-in macro incf is an abbreviation for setf of +. Instead of

```
(setf x (+ x y))
```

we say just

```
(incf x y)
```

The new _f is to be a generalization of this idea: while incf expands into a call to +, _f will expand into a call to the operator given as the first argument. For example, in the definition of scale-objs on page 115, we had to write

```
(setf (obj-dx o) (* (obj-dx o) factor))
```

With _f this will become

```
(_f * (obj-dx o) factor)
```

The incorrect way to write _f would be:

```
(defmacro _f (op place &rest args)                    ; wrong
  `(setf ,place (,op ,place ,@args)))
```

Unfortunately, we can't define a correct _f with define-modify-macro, because the operator to be applied to the generalized variable is given as an argument.

More complex macros like this one have to be written by hand. To make such macros easier to write, Common Lisp provides the function get-setf-method, which takes a generalized variable and returns all the information necessary to retrieve or set its value. We will see how to use this information by hand-generating an expansion for:

```
(incf (aref a (incf i)))
```

When we call get-setf-method on the generalized variable, we get five values intended for use as ingredients in the macroexpansion:

```
> (get-setf-method '(aref a (incf i)))
(#:G4 #:G5)
(A (INCF I))
(#:G6)
(SYSTEM:SET-AREF #:G6 #:G4 #:G5)
(AREF #:G4 #:G5)
```

The first two values are lists of temporary variables and the values that should be assigned to them. So we can begin the expansion with:

```
(let* ((#:g4 a)
       (#:g5 (incf i)))
  ...)
```

These bindings should be created in a `let*` because in the general case the value forms can refer to earlier variables. The third[3] and fifth values are another temporary variable and the form that will return the original value of the generalized variable. Since we want to add `1` to this value, we wrap the latter in a call to `1+`:

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  ...)
```

Finally, the fourth value returned by `get-setf-method` is the assignment that must be made within the scope of the new bindings:

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  (system:set-aref #:g6 #:g4 #:g5))
```

More often than not, this form will refer to internal functions which are not part of Common Lisp. Usually `setf` masks the presence of these functions, but they have to exist somewhere. Everything about them is implementation-dependent, so portable code should use forms returned by `get-setf-method`, rather than referring directly to functions like `system:set-aref`.

Now to implement `_f` we write a macro which does almost exactly what we did when expanding `incf` by hand. The only difference is that, instead of wrapping the last form in the `let*` in a call to `1+`, we wrap it in an expression made from the arguments to `_f`. The definition of `_f` is shown in Figure 12.3.

[3]The third value is currently always a list of one element. It is returned as a list to provide the (so far unconsummated) potential to store multiple values in generalized variables.

```
(defmacro _f (op place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    `(let* (,@(mapcar #'list vars forms)
            (,(car var) (,op ,access ,@args)))
       ,set)))

(defmacro pull (obj place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete ,g ,access ,@args)))
         ,set))))

(defmacro pull-if (test place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    (let ((g (gensym)))
      `(let* ((,g ,test)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete-if ,g ,access ,@args)))
         ,set))))

(defmacro popn (n place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
    (with-gensyms (gn glst)
      `(let* ((,gn ,n)
              ,@(mapcar #'list vars forms)
              (,glst ,access)
              (,(car var) (nthcdr ,gn ,glst)))
         (prog1 (subseq ,glst 0 ,gn)
                ,set)))))
```

Figure 12.3: More complex macros on `setf`.

This utility is quite a useful one. Now that we have it, for example, we can easily replace any named function with a memoized (Section 5.3) equivalent.[4] To memoize `foo` we would say:

```
(_f memoize (symbol-function 'foo))
```

Having `_f` also makes it easy to define other macros on `setf`. For example, we could now define `conc1f` (Figure 12.2) as:

```
(defmacro conc1f (lst obj)
  `(_f nconc ,lst (list ,obj)))
```

Figure 12.3 contains some other useful macros on `setf`. The next, `pull`, is intended as a complement to the built-in `pushnew`. The pair are like more discerning versions of `push` and `pop`; `pushnew` pushes a new element onto a list if it is not already a member, and `pull` destructively removes selected elements from a list. The `&rest` parameter in `pull`'s definition makes `pull` able to accept all the same keyword parameters as `delete`:

```
> (setq x '(1 2 (a b) 3))
(1 2 (A B) 3)
> (pull 2 x)
(1 (A B) 3)
> (pull '(a b) x :test #'equal)
(1 3)
> x
(1 3)
```

You could almost think of this macro as if it were defined:

```
(defmacro pull (obj seq &rest args)                    ; wrong
  `(setf ,seq (delete ,obj ,seq ,@args)))
```

though if it really were defined that way, it would be subject to problems with both order and number of evaluations. We could define a version of `pull` as a simple modify-macro:

```
(define-modify-macro pull (obj &rest args)
  (lambda (seq obj &rest args)
    (apply #'delete obj seq args)))
```

[4]Built-in functions should not be memoized in this way, though. Common Lisp forbids the redefinition of built-in functions.

but since modify-macros must take the generalized variable as their first argument, we would have to give the first two arguments in reverse order, which would be less intuitive.

The more general `pull-if` takes an initial function argument, and expands into a `delete-if` instead of a `delete`:

```
> (let ((lst '(1 2 3 4 5 6)))
    (pull-if #'oddp lst)
    lst)
(2 4 6)
```

These two macros illustrate another general point. If the underlying function takes optional arguments, so should the macro built upon it. Both `pull` and `pull-if` pass optional arguments on to their `delete`s.

The final macro in Figure 12.3, `popn`, is a generalization of `pop`. Instead of popping just one element of a list, it pops and returns a subsequence of arbitrary length:

```
> (setq x '(a b c d e f))
(A B C D E F)
> (popn 3 x)
(A B C)
> x
(D E F)
```

Figure 12.4 contains a macro which sorts its arguments. If `x` and `y` are variables and we want to ensure that `x` does not have the lower of the two values, we can write:

```
(if (> y x) (rotatef x y))
```

But if we want to do this for three or more variables, the code required grows rapidly. Instead of writing it by hand, we can have `sortf` write it for us. This macro takes a comparison operator plus any number of generalized variables, and swaps their values until they are in the order dictated by the operator. In the simplest case, the arguments could be ordinary variables:

```
> (setq x 1 y 2 z 3)
3
> (sortf > x y z)
3
> (list x y z)
(3 2 1)
```

```
(defmacro sortf (op &rest places)
  (let* ((meths (mapcar #'(lambda (p)
                            (multiple-value-list
                              (get-setf-method p)))
                        places))
         (temps (apply #'append (mapcar #'third meths))))
    `(let* ,(mapcar #'list
                    (mapcan #'(lambda (m)
                                (append (first m)
                                        (third m)))
                            meths)
                    (mapcan #'(lambda (m)
                                (append (second m)
                                        (list (fifth m))))
                            meths))
       ,@(mapcon #'(lambda (rest)
                     (mapcar
                       #'(lambda (arg)
                           `(unless (,op ,(car rest) ,arg)
                              (rotatef ,(car rest) ,arg)))
                       (cdr rest)))
                 temps)
       ,@(mapcar #'fourth meths))))
```

Figure 12.4: A macro which sorts its arguments.

In general, they could be any invertible expressions. Suppose `cake` is an invertible function which returns someone's piece of cake, and `bigger` is a comparison function defined on pieces of cake. If we want to enforce the rule that the `cake` of `moe` is no less than the `cake` of `larry`, which is no less than that of `curly`, we write:

```
(sortf bigger (cake 'moe) (cake 'larry) (cake 'curly))
```

The definition of `sortf` is similar in outline to that of `_f`. It begins with a `let*` in which the temporary variables returned by `get-setf-method` are bound to the initial values of the generalized variables. The core of `sortf` is the central `mapcon` expression, which generates code to sort these temporary variables. The code generated by this portion of the macro grows exponentially with the number of arguments. After sorting, the generalized variables are reassigned using the

```
(sortf > x (aref ar (incf i)) (car lst))
```

expands (in one possible implementation) into:

```
(let* ((#:g1 x)
       (#:g4 ar)
       (#:g3 (incf i))
       (#:g2 (aref #:g4 #:g3))
       (#:g6 lst)
       (#:g5 (car #:g6)))
  (unless (> #:g1 #:g2)
    (rotatef #:g1 #:g2))
  (unless (> #:g1 #:g5)
    (rotatef #:g1 #:g5))
  (unless (> #:g2 #:g5)
    (rotatef #:g2 #:g5))
  (setq x #:g1)
  (system:set-aref #:g2 #:g4 #:g3)
  (system:set-car #:g6 #:g5))
```

Figure 12.5: Expansion of a call to `sortf`.

forms returned by `get-setf-method`. The algorithm used is the $O(n^2)$ bubble-sort, but this macro is not intended to be called with huge numbers of arguments.

Figure 12.5 shows the expansion of a call to `sortf`. In the initial `let*`, the arguments and their subforms are carefully evaluated in left-to-right order. Then appear three expressions which compare and possibly swap the values of the temporary variables: the first is compared to the second, then the first to the third, then the second to the third. Finally the the generalized variables are reassigned left-to-right. Although the issue rarely arises, macro arguments should usually be assigned left-to-right, as well as being evaluated in this order.

Operators like `_f` and `sortf` bear a certain resemblance to functions that take functional arguments. It should be understood that they are something quite different. A function like `find-if` takes a function and calls it; a macro like `_f` takes a *name*, and makes it the car of an expression. Both `_f` and `sortf` could have been written to take functional arguments. For example, `_f` could have been written:

```
(defmacro _f (op place &rest args)
  (let ((g (gensym)))
    (multiple-value-bind (vars forms var set access)
                         (get-setf-method place)
    `(let* ((,g ,op)
            ,@(mapcar #'list vars forms)
            (,(car var) (funcall ,g ,access ,@args)))
       ,set))))
```

and called `(_f #'+ x 1)`. But the original version of `_f` can do anything this one could, and since it deals in names, it can also take the name of a macro or special form. As well as `+`, you could call, for example, `nif` (page 150):

```
> (let ((x 2))
    (_f nif x 'p 'z 'n)
    x)
P
```

12.5 Defining Inversions

Section 12.1 explained that any macro call which expands into an invertible reference will itself be invertible. You don't have to define operators as macros just to make them invertible, though. By using `defsetf` you can tell Lisp how to invert any function or macro call.

This macro can be used in two ways. In the simplest case, its arguments are two symbols:

```
(defsetf symbol-value set)
```

In the more complicated form, a call to `defsetf` is like a call to `defmacro`, with an additional parameter for the updated value form. For example, this would define a possible inversion for `car`:

```
(defsetf car (lst) (new-car)
  `(progn (rplaca ,lst ,new-car)
          ,new-car))
```

There is one important difference between `defmacro` and `defsetf`: the latter automatically creates gensyms for its arguments. With the definition given above, `(setf (car x) y)` would expand into:

```
(let* ((#:g2 x)
       (#:g1 y))
  (progn (rplaca #:g2 #:g1)
         #:g1))
```

```
(defvar *cache* (make-hash-table))

(defun retrieve (key)
  (multiple-value-bind (x y) (gethash key *cache*)
    (if y
        (values x y)
        (cdr (assoc key *world*)))))

(defsetf retrieve (key) (val)
  `(setf (gethash ,key *cache*) ,val))
```

Figure 12.6: An asymmetric inversion.

Thus we can write `defsetf` expanders without having to worry about variable capture, or number or order of evaluations.

In CLTL2 Common Lisp, it is possible to define `setf` inversions directly with `defun`, so the previous example could also be written:

```
(defun (setf car) (new-car lst)
  (rplaca lst new-car)
  new-car)
```

The updated value should be the first parameter in such a function. It is also conventional to return this value as the value of the function.

The examples so far have suggested that generalized variables are supposed to refer to a place in a data structure. The villain carries his hostage down to the dungeon, and the rescuing hero carries her back up again; they both follow the same path, but in different directions. It's not surprising if people have the impression that `setf` must work this way, because all the predefined inversions seem to be of this form; indeed, `place` is the conventional name for a parameter which is to be inverted.

In principle, `setf` is more general: an access form and its inversion need not even operate on the same data structure. Suppose that in some application we want to cache database updates. This could be necessary, for example, if it were not efficient to do real updates on the fly, or if all the updates had to be verified for consistency before committing to them.

Suppose that `*world*` is the actual database. For simplicity, we will make it an assoc-list whose elements are of the form (*key* . *val*). Figure 12.6 shows a lookup function called `retrieve`. If `*world*` is

```
((a . 2) (b . 16) (c . 50) (d . 20) (f . 12))
```

then

```
> (retrieve 'c)
50
```

Unlike a call to `car`, a call to `retrieve` does not refer to a specific place in a data structure. The return value could come from one of two places. And the inversion of retrieve, also defined in Figure 12.6, only refers to one of them:

```
> (setf (retrieve 'n) 77)
77
> (retrieve 'n)
77
T
```

This lookup returns a second value of `t`, indicating that the answer was found in the cache.

Like macros themselves, generalized variables are an abstraction of remarkable power. There is probably more to be discovered here. Certainly individual users are likely to discover ways in which the use of generalized variables could lead to more elegant or more powerful programs. But it may also be possible to use `setf` inversion in new ways, or to discover other classes of similarly useful ◦ transformations.

13

Computation at Compile-Time

The preceding chapters described several types of operators which have to be implemented by macros. This one describes a class of problems which could be solved by functions, but where macros are more efficient. Section 8.2 listed the pros and cons of using macros in a given situation. Among the pros was "computation at compile-time." By defining an operator as a macro, you can sometimes make it do some of its work when it is expanded. This chapter looks at macros which take advantage of this possibility.

13.1 New Utilities

Section 8.2 raised the possibility of using macros to shift computation to compile-time. There we had as an example the macro `avg`, which returns the average of its arguments:

```
> (avg pi 4 5)
4.047...
```

Figure 13.1 shows `avg` defined first as a function and then as a macro. When `avg` is defined as a macro, the call to `length` can be made at compile-time. In the macro version we also avoid the expense of manipulating the `&rest` parameter at runtime. In many implementations, `avg` will be faster written as a macro.

The kind of savings which comes from knowing the number of arguments at expansion-time can be combined with the kind we get from `in` (page 152), where it was possible to avoid even evaluating some of the arguments. Figure 13.2 contains two versions of `most-of`, which returns true if most of its arguments do:

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

Figure 13.1: Shifting computation when finding averages.

```
(defun most-of (&rest args)
  (let ((all 0)
        (hits 0))
    (dolist (a args)
      (incf all)
      (if a (incf hits)))
    (> hits (/ all 2))))

(defmacro most-of (&rest args)
  (let ((need (floor (/ (length args) 2)))
        (hits (gensym)))
    `(let ((,hits 0))
       (or ,@(mapcar #'(lambda (a)
                         `(and ,a (> (incf ,hits) ,need)))
                     args)))))
```

Figure 13.2: Shifting and avoiding computation.

```
> (most-of t t t nil)
T
```

The macro version expands into code which, like `in`, only evaluates as many of the arguments as it needs to. For example, `(most-of (a) (b) (c))` expands into the equivalent of:

```
(let ((count 0))
  (or (and (a) (> (incf count) 1))
      (and (b) (> (incf count) 1))
      (and (c) (> (incf count) 1))))
```

In the best case, just over half the arguments will be evaluated.

```
(defun nthmost (n lst)
  (nth n (sort (copy-list lst) #'>)))

(defmacro nthmost (n lst)
  (if (and (integerp n) (< n 20))
      (with-gensyms (glst gi)
        (let ((syms (map0-n #'(lambda (x) (gensym)) n)))
          `(let ((,glst ,lst))
             (unless (< (length ,glst) ,(1+ n))
               ,@(gen-start glst syms)
               (dolist (,gi ,glst)
                 ,(nthmost-gen gi syms t))
               ,(car (last syms))))))
      `(nth ,n (sort (copy-list ,lst) #'>))))

(defun gen-start (glst syms)
  (reverse
    (maplist #'(lambda (syms)
                 (let ((var (gensym)))
                   `(let ((,var (pop ,glst)))
                      ,(nthmost-gen var (reverse syms)))))
             (reverse syms))))

(defun nthmost-gen (var vars &optional long?)
  (if (null vars)
      nil
      (let ((else (nthmost-gen var (cdr vars) long?)))
        (if (and (not long?) (null else))
            `(setq ,(car vars) ,var)
            `(if (> ,var ,(car vars))
                 (setq ,@(mapcan #'list
                                 (reverse vars)
                                 (cdr (reverse vars)))
                       ,(car vars) ,var)
                 ,else)))))
```

Figure 13.3: Use of arguments known at compile-time.

A macro may also be able to shift computation to compile-time if the values of particular arguments are known. Figure 13.3 contains an example of such a macro. The function `nthmost` takes a number *n* and a list of numbers, and returns the *n*th largest among them; like other sequence functions, it is zero-indexed:

```
> (nthmost 2 '(2 6 1 5 3 4))
4
```

The function version is written very simply. It sorts the list and calls `nth` on the result. Since `sort` is destructive, `nthmost` copies the list before sorting it. Written thus, `nthmost` is inefficient is two respects: it conses, and it sorts the entire list of arguments, though all we care about are the top *n*.

If we know *n* at compile-time, we can approach the problem differently. The rest of the code in Figure 13.3 defines a macro version of `nthmost`. The first thing this macro does is look at its first argument. If the first argument is not a literal number, it expands into the same code we saw above. If the first argument is a number, we can follow a different course. If you wanted to find, say, the third biggest cookie on a plate, you could do it by looking at each cookie in turn, always keeping in your hand the three biggest found so far. When you have looked at all the cookies, the smallest cookie in your hand is the one you are looking for. If *n* is a small constant, not proportional to the number of cookies, then this technique gets you a given cookie with less effort that it would take to sort all of them first.

This is the strategy followed when *n* is known at expansion-time. In its expansion, the macro creates *n* variables, then calls `nthmost-gen` to generate the code which has to be evaluated upon looking at each cookie. Figure 13.4 shows a sample macroexpansion. The macro `nthmost` behaves just like the original function, except that it can't be passed as an argument to `apply`. The justification for using a macro is purely one of efficiency: the macro version does not cons at runtime, and if *n* is a small constant, performs fewer comparisons.

To have efficient programs, must one then take the trouble to write such huge macros? In this case, probably not. The two versions of `nthmost` are intended as an example of a general principle: when some arguments are known at compile-time, you can use a macro to generate more efficient code. Whether or not you take advantage of this possibility will depend on how much you stand to gain, and how much more effort it will take to write an efficient macro version. Since the macro version of `nthmost` is long and complicated, it would only be worth writing in extreme cases. However, information known at compile-time is always a factor worth considering, even if you choose not to take advantage of it.

```
(nthmost 2 nums)
```

expands into:

```
(let ((#:g7 nums))
  (unless (< (length #:g7) 3)
    (let ((#:g6 (pop #:g7)))
      (setq #:g1 #:g6))
    (let ((#:g5 (pop #:g7)))
      (if (> #:g5 #:g1)
          (setq #:g2 #:g1 #:g1 #:g5)
          (setq #:g2 #:g5)))
    (let ((#:g4 (pop #:g7)))
      (if (> #:g4 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g4)
          (if (> #:g4 #:g2)
              (setq #:g3 #:g2 #:g2 #:g4)
              (setq #:g3 #:g4))))
    (dolist (#:g8 #:g7)
      (if (> #:g8 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g8)
          (if (> #:g8 #:g2)
              (setq #:g3 #:g2 #:g2 #:g8)
              (if (> #:g8 #:g3)
                  (setq #:g3 #:g8)
                  nil))))
    #:g3))
```

Figure 13.4: Expansion of `nthmost`.

13.2 Example: Bezier Curves

Like the `with-` macro (Section 11.2), the macro for computation at compile-time is more likely to be written for a specific application than as a general-purpose utility. How much can a general-purpose utility know at compile-time? The number of arguments it has been given, and perhaps some of their values. If we want to use other constraints, they will probably have to be ones imposed by individual programs.

As an example, this section shows how macros can speed up the generation of Bezier curves. Curves must be generated fast if they are being manipulated interactively. It turns out that if the number of segments in the curve is known beforehand, most of the computation can be done at compile-time. By writing our curve-generator as a macro, we can weave precomputed values right into code. This should be even faster than the more usual optimization of storing them in an array.

A Bezier curve is defined in terms of four points—two endpoints and two control points. When we are working in two dimensions, these points define parametric equations for the x and y coordinates of points on the curve. If the two endpoints are (x_0, y_0) and (x_3, y_3) and the two control points are (x_1, y_1) and (x_2, y_2), then the equations defining points on the curve are:

$$x = (x_3 - 3x_2 + 3x_1 - x_0)u^3 + (3x_2 - 6x_1 + 3x_0)u^2 + (3x_1 - 3x_0)u + x_0$$

$$y = (y_3 - 3y_2 + 3y_1 - y_0)u^3 + (3y_2 - 6y_1 + 3y_0)u^2 + (3y_1 - 3y_0)u + y_0$$

If we evaluate these equations for n values of u between 0 and 1, we get n points on the curve. For example, if we want to draw the curve as 20 segments, then we would evaluate the equations for $u = .05, .1, \ldots, .95$. There is no need to evaluate them for u of 0 or 1, because if $u = 0$ they will yield the first endpoint (x_0, y_0), and if $u = 1$ they will yield the second endpoint (x_3, y_3).

An obvious optimization is to make n fixed, calculate the powers of u beforehand, and store them in an $(n-1) \times 3$ array. By defining the curve-generator as a macro, we can do even better. If n is known at expansion-time, the program could simply expand into n line-drawing commands. The precomputed powers of u, instead of being stored in an array, could be inserted as literal values right into the macro expansion.

Figure 13.5 contains a curve-generating macro which implements this strategy. Instead of drawing lines immediately, it dumps the generated points into an array. When a curve is moving interactively, each instance has to be drawn twice: once to show it, and again to erase it before drawing the next. In the meantime, the points have to be saved somewhere.

With $n = 20$, `genbez` expands into 21 `setf`s. Since the powers of u appear directly in the code, we save the cost of looking them up at runtime, and the cost of computing them at startup. Like the powers of u, the array indices appear as constants in the expansion, so the bounds-checking for the `(setf aref)`s could also be done at compile-time.

13.3 Applications

Later chapters contain several other macros which use information available at compile-time. A good example is `if-match` (page 242). Pattern-matchers compare two sequences, possibly containing variables, to see if there is some way of assigning values to the variables which will make the two sequences equal. The

```
(defconstant *segs* 20)
(defconstant *du*   (/ 1.0 *segs*))
(defconstant *pts*  (make-array (list (1+ *segs*) 2)))

(defmacro genbez (x0 y0 x1 y1 x2 y2 x3 y3)
  (with-gensyms (gx0 gx1 gy0 gy1 gx3 gy3)
    `(let ((,gx0 ,x0) (,gy0 ,y0)
           (,gx1 ,x1) (,gy1 ,y1)
           (,gx3 ,x3) (,gy3 ,y3))
       (let ((cx (* (- ,gx1 ,gx0) 3))
             (cy (* (- ,gy1 ,gy0) 3))
             (px (* (- ,x2 ,gx1) 3))
             (py (* (- ,y2 ,gy1) 3)))
         (let ((bx (- px cx))
               (by (- py cy))
               (ax (- ,gx3 px ,gx0))
               (ay (- ,gy3 py ,gy0)))
           (setf (aref *pts* 0 0) ,gx0
                 (aref *pts* 0 1) ,gy0)
           ,@(map1-n #'(lambda (n)
                         (let* ((u (* n *du*))
                                (u^2 (* u u))
                                (u^3 (expt u 3)))
                           `(setf (aref *pts* ,n 0)
                                  (+ (* ax ,u^3)
                                     (* bx ,u^2)
                                     (* cx ,u)
                                     ,gx0)
                                  (aref *pts* ,n 1)
                                  (+ (* ay ,u^3)
                                     (* by ,u^2)
                                     (* cy ,u)
                                     ,gy0))))
                     (1- *segs*))
             (setf (aref *pts* *segs* 0) ,gx3
                   (aref *pts* *segs* 1) ,gy3))))))
```

Figure 13.5: Macro for generating Bezier curves.

design of `if-match` shows that if one of the sequences is known at compile-time, and only that one contains variables, then matching can be done more efficiently. Instead of comparing the two sequences at runtime and consing up lists to hold the variable bindings established in the process, we can have a macro generate code to perform the exact comparisons dictated by the known sequence, and can store the bindings in real Lisp variables.

The embedded languages described in Chapters 19–24 also, for the most part, take advantage of information available at compile-time. Since an embedded language is a compiler of sorts, it's only natural that it should use such information. As a general rule, the more elaborate the macro, the more constraints it imposes on its arguments, and the better your chances of using these constraints to generate efficient code.

14

Anaphoric Macros

Chapter 9 treated variable capture exclusively as a problem—as something which happens inadvertently, and which can only affect programs for the worse. This chapter will show that variable capture can also be used constructively. There are some useful macros which couldn't be written without it.

It's not uncommon in a Lisp program to want to test whether an expression returns a non-nil value, and if so, to do something with the value. If the expression is costly to evaluate, then one must normally do something like this:

```
(let ((result (big-long-calculation)))
  (if result
      (foo result)))
```

Wouldn't it be easier if we could just say, as we would in English:

```
(if (big-long-calculation)
    (foo it))
```

By taking advantage of variable capture, we can write a version of `if` which works just this way.

14.1 Anaphoric Variants

In natural language, an *anaphor* is an expression which refers back in the conversation. The most common anaphor in English is probably "it," as in "Get the wrench and put it on the table." Anaphora are a great convenience in everyday language—imagine trying to get along without them—but they don't appear much in programming languages. For the most part, this is good. Anaphoric expressions are often genuinely ambiguous, and present-day programming languages are not designed to handle ambiguity.

However, it is possible to introduce a very limited form of anaphora into Lisp programs without causing ambiguity. An anaphor, it turns out, is a lot like a captured symbol. We can use anaphora in programs by designating certain symbols to serve as pronouns, and then writing macros intentionally to capture these symbols.

In the new version of `if`, the symbol `it` is the one we want to capture. The anaphoric if, called `aif` for short, is defined as follows:

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

and used as in the previous example:

```
(aif (big-long-calculation)
     (foo it))
```

When you use an `aif`, the symbol `it` is left bound to the result returned by the test clause. In the macro call, `it` seems to be free, but in fact the expression `(foo it)` will be inserted by expansion of the `aif` into a context in which the symbol `it` is bound:

```
(let ((it (big-long-calculation)))
  (if it (foo it) nil))
```

So a symbol which looks free in the source code is left bound by the macroexpansion. All the anaphoric macros in this chapter use some variation of the same technique.

Figure 14.1 contains anaphoric variants of several Common Lisp operators. After `aif` comes `awhen`, the obvious anaphoric variant of `when`:

```
(awhen (big-long-calculation)
  (foo it)
  (bar it))
```

Both `aif` and `awhen` are frequently useful, but `awhile` is probably unique among the anaphoric macros in being more often needed than its regular cousin, `while` (defined on page 91). Macros like `while` and `awhile` are typically used in situations where a program needs to poll some outside source. And when you are polling a source, unless you are simply waiting for it to change state, you will usually want to do something with the object you find there:

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
  `(aif ,test-form
        (progn ,@body)))

(defmacro awhile (expr &body body)
  `(do ((it ,expr ,expr))
       ((not it))
     ,@body))

(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (sym (gensym)))
        `(let ((,sym ,(car cl1)))
           (if ,sym
               (let ((it ,sym)) ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

Figure 14.1: Anaphoric variants of Common Lisp operators.

```
(awhile (poll *fridge*)
  (eat it))
```

The definition of aand is a bit more complicated than the preceding ones. It provides an anaphoric version of and; during the evaluation of each of its arguments, it will be bound to the value returned by the previous argument.[1] In practice, aand tends to be used in programs which make conditional queries, as in:

[1]Although one tends to think of and and or together, there would be no point in writing an anaphoric version of or. An argument in an or expression is evaluated only if the previous argument evaluated to nil, so there would be nothing useful for an anaphor to refer to in an aor.

```
(aand (owner x) (address it) (town it))
```

which returns the town (if there is one) of the address (if there is one) of the owner (if there is one) of `x`. Without `aand`, this expression would have to be written

```
(let ((own (owner x)))
  (if own
      (let ((adr (address own)))
        (if adr (town adr)))))
```

The definition of `aand` shows that the expansion will vary depending on the number of arguments in the macro call. If there are no arguments, then `aand`, like the regular `and`, should simply return `t`. Otherwise the expansion is generated recursively, each step yielding one layer in a chain of nested `aif`s:

```
(aif ⟨first argument⟩
     ⟨expansion for rest of arguments⟩)
```

The expansion of an `aand` must terminate when there is one argument left, instead of working its way down to `nil` like most recursive functions. If the recursion continued until no conjuncts remained, the expansion would always be of the form:

```
(aif ⟨c1⟩
     ⋮
     (aif ⟨cn⟩
          t)...)
```

Such an expression would always return `t` or `nil`, and the example above wouldn't work as intended.

Section 10.4 warned that if a macro always yielded an expansion containing a call to itself, the expansion would never terminate. Though recursive, `aand` is safe because in the base case its expansion doesn't refer to `aand`.

The last example, `acond`, is meant for those cases where the remainder of a `cond` clause wants to use the value returned by the test expression. (This situation arises so often that some Scheme implementations provide a way to use the value returned by the test expression in a `cond` clause.)

In the expansion of an `acond` clause, the result of the test expression will initially be kept in a gensymed variable, in order that the symbol `it` may be bound only within the remainder of the clause. When macros create bindings, they should always do so over the narrowest possible scope. Here, if we dispensed with the

```
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))

(defmacro ablock (tag &rest args)
  `(block ,tag
     ,(funcall (alambda (args)
                 (case (length args)
                   (0 nil)
                   (1 (car args))
                   (t `(let ((it ,(car args)))
                         ,(self (cdr args))))))
               args)))
```

Figure 14.2: More anaphoric variants.

gensym and instead bound `it` immediately to the result of the test expression, as in:

```
(defmacro acond (&rest clauses)                          ; wrong
  (if (null clauses)
      nil
      (let ((cl1 (car clauses)))
        `(let ((it ,(car cl1)))
           (if it
               (progn ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

then that binding of `it` would also have within its scope the *following* test expression.

Figure 14.2 contains some more complicated anaphoric variants. The macro `alambda` is for referring literally to recursive functions. When does one want to refer literally to a recursive function? We can refer literally to a function by using a sharp-quoted lambda-expression:

```
#'(lambda (x) (* x 2))
```

But as Chapter 2 explained, you can't express a recursive function with a simple lambda-expression. Instead you have to define a local function with `labels`. The following function (reproduced from page 22)

```
(defun count-instances (obj lists)
  (labels ((instances-in (list)
             (if list
                 (+ (if (eq (car list) obj) 1 0)
                    (instances-in (cdr list)))
                 0)))
    (mapcar #'instances-in lists)))
```

takes an object and a list, and returns a list of the number of occurrences of the object in each element:

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

With anaphora we can make what amounts to a literal recursive function. The `alambda` macro uses `labels` to create one, and thus can be used to express, for example, the factorial function:

```
(alambda (x) (if (= x 0) 1 (* x (self (1- x)))))
```

Using `alambda` we can define an equivalent version of `count-instances` as follows:

```
(defun count-instances (obj lists)
  (mapcar (alambda (list)
            (if list
                (+ (if (eq (car list) obj) 1 0)
                   (self (cdr list)))
                0))
          lists))
```

Unlike the other macros in Figures 14.1 and 14.2, which all capture `it`, `alambda` captures `self`. An instance of `alambda` expands into a `labels` expression in which `self` is bound to the function being defined. As well as being smaller, `alambda` expressions look like familiar `lambda` expressions, making code which uses them easier to read.

The new macro is used in the definition of `ablock`, an anaphoric version of the built-in `block` special form. In a `block`, the arguments are evaluated left-to-right. The same happens in an `ablock`, but within each the variable `it` will be bound to the value of the previous expression.

This macro should be used with discretion. Though convenient at times, `ablock` would tend to beat what could be nice functional programs into imperative form. The following is, unfortunately, a characteristically ugly example:

```
> (ablock north-pole
    (princ "ho ")
    (princ it)
    (princ it)
    (return-from north-pole))
ho ho ho
NIL
```

Whenever a macro which does intentional variable capture is exported to another package, it is necessary also to export the symbol being captured. For example, wherever `aif` is exported, `it` should be as well. Otherwise the `it` which appears in the macro definition would be a different symbol from an `it` used in a macro call.

14.2 Failure

In Common Lisp the symbol `nil` has at least three different jobs. It is first of all the empty list, so that

```
> (cdr '(a))
NIL
```

As well as the empty list, `nil` is used to represent falsity, as in

```
> (= 1 0)
NIL
```

And finally, functions return `nil` to indicate failure. For example, the job of the built-in `find-if` is to return the first element of a list which satisfies some test. If no such element is found, `find-if` returns `nil`:

```
> (find-if #'oddp '(2 4 6))
NIL
```

Unfortunately, we can't tell this case from the one in which `find-if` succeeds, but succeeds in finding `nil`:

```
> (find-if #'null '(2 nil 6))
NIL
```

In practice, it doesn't cause too much trouble to use `nil` to represent both falsity and the empty list. In fact, it can be rather convenient. However, it is a pain to have `nil` represent failure as well, because it means that the result returned by a function like `find-if` can be ambiguous.

The problem of distinguishing between failure and a `nil` return value arises with any function which looks things up. Common Lisp offers no less than three solutions to the problem. The most common approach, before multiple return values, was to return gratuitous list structure. There is no trouble distinguishing failure with `assoc`, for example; when successful it returns the whole pair in question:

```
> (setq synonyms '((yes . t) (no . nil)))
((YES . T) (NO))
> (assoc 'no synonyms)
(NO)
```

Following this approach, if we were worried about ambiguity with `find-if`, we would use `member-if`, which instead of just returning the element satisfying the test, returns the whole cdr which begins with it:

```
> (member-if #'null '(2 nil 6))
(NIL 6)
```

Since the advent of multiple return values, there has been another solution to this problem: use one value for data and a second to indicate success or failure. The built-in `gethash` works this way. It always returns two values, the second indicating whether anything was found:

```
> (setf edible                       (make-hash-table)
        (gethash 'olive-oil edible) t
        (gethash 'motor-oil edible) nil)
NIL
> (gethash 'motor-oil edible)
NIL
T
```

So if you want to detect all three possible cases, you can use an idiom like the following:

```
(defun edible? (x)
  (multiple-value-bind (val found?) (gethash x edible)
    (if found?
        (if val 'yes 'no)
        'maybe)))
```

thereby distinguishing falsity from failure:

```
> (mapcar #'edible? '(motor-oil olive-oil iguana))
(NO YES MAYBE)
```

Common Lisp supports yet a third way of indicating failure: to have the access function take as an argument a special object, presumably a gensym, to be returned in case of failure. This approach is used with `get`, which takes an optional argument saying what to return if the specified property isn't found:

```
> (get 'life 'meaning (gensym))
#:G618
```

Where multiple return values are possible, the approach used by `gethash` is the cleanest. We don't want to have to pass additional arguments to every access function, as we do with `get`. And between the other two alternatives, using multiple values is the more general; `find-if` could be written to return two values, but `gethash` could not, without consing, be written to return disambiguating list structure. Thus in writing new functions for lookup, or for other tasks where failure is possible, it will usually be better to follow the model of `gethash`.

The idiom found in `edible?` is just the sort of bookkeeping which is well hidden by a macro. For access functions like `gethash` we will want a new version of `aif` which, instead of binding and testing the same value, binds the first but also tests the second. The new version of `aif`, called `aif2`, is shown in Figure 14.3. Using it we could write `edible?` as:

```
(defun edible? (x)
  (aif2 (gethash x edible)
        (if it 'yes 'no)
        'maybe))
```

Figure 14.3 also contains similarly altered versions of `awhen`, `awhile`, and `acond`. For an example of the use of `acond2`, see the definition of `match` on page 239. By using this macro we are able to express in the form of a `cond` a function that would otherwise be much longer and less symmetrical.

The built-in `read` indicates failure in the same way as `get`. It takes optional arguments saying whether or not to generate an error in case of eof, and if not, what value to return. Figure 14.4 contains an alternative version of `read` which uses a second return value to indicate failure: `read2` returns two values, the input expression and a flag which is `nil` upon eof. It calls `read` with a gensym to be returned in case of eof, but to save the trouble of building the gensym each time `read2` is called, the function is defined as a closure with a private copy of a gensym made at compile time.

Figure 14.4 also contains a convenient macro to iterate over the expressions in a file, written using `awhile2` and `read2`. Using `do-file` we could, for example, write a version of `load` as:

```
(defun our-load (filename)
  (do-file filename (eval it)))
```

```
(defmacro aif2 (test &optional then else)
  (let ((win (gensym)))
    `(multiple-value-bind (it ,win) ,test
       (if (or it ,win) ,then ,else))))

(defmacro awhen2 (test &body body)
  `(aif2 ,test
         (progn ,@body)))

(defmacro awhile2 (test &body body)
  (let ((flag (gensym)))
    `(let ((,flag t))
       (while ,flag
         (aif2 ,test
               (progn ,@body)
               (setq ,flag nil))))))

(defmacro acond2 (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (val (gensym))
            (win (gensym)))
        `(multiple-value-bind (,val ,win) ,(car cl1)
           (if (or ,val ,win)
               (let ((it ,val)) ,@(cdr cl1))
               (acond2 ,@(cdr clauses)))))))
```

Figure 14.3: Multiple-value anaphoric macros.

14.3 Referential Transparency

Anaphoric macros are sometimes said to violate referential transparency, which Gelernter and Jagannathan define as follows:

> A language is *referentially transparent* if (a) every subexpression can be replaced by any other that's equal to it in value and (b) all occurrences of an expression within a given context yield the same value.

Note that this standard applies to languages, not to programs. No language with assignment is referentially transparent. The first and the last x in this expression

```
(let ((g (gensym)))
  (defun read2 (&optional (str *standard-input*))
    (let ((val (read str nil g)))
      (unless (equal val g) (values val t)))))

(defmacro do-file (filename &body body)
  (let ((str (gensym)))
    `(with-open-file (,str ,filename)
       (awhile2 (read2 ,str)
         ,@body))))
```

Figure 14.4: File utilities.

```
(list x
      (setq x (not x))
      x)
```

yield different values, because a `setq` intervenes. Admittedly, this is ugly code. The fact that it is even possible means that Lisp is not referentially transparent.

Norvig mentions that it would be convenient to redefine `if` as: ○

```
(defmacro if (test then &optional else)
  `(let ((that ,test))
     (if that ,then ,else)))
```

but rejects this macro on the grounds that it violates referential transparency.

However, the problem here comes from redefining built-in operators, not from using anaphora. Clause (b) of the definition above requires that an expression always return the same value "within a given context." It is no problem if, within this `let` expression,

```
(let ((that 'which))
  ...)
```

the symbol `that` denotes a new variable, because `let` is advertised to create a new context.

The trouble with the macro above is that it redefines `if`, which is *not* supposed to create a new context. This problem goes away if we give anaphoric macros distinct names. (As of CLTL2, it is illegal to redefine `if` anyway.) As long as it is part of the definition of `aif` that it establishes a new context in which `it` is a new variable, such a macro does not violate referential transparency.

Now, `aif` does violate another convention, which has nothing to do with referential transparency: that newly established variables somehow be indicated in the source code. The `let` expression above clearly indicates that `that` will refer to a new variable. It could be argued that the binding of `it` within an `aif` is not so clear. However, this is not a very strong argument: `aif` only creates one variable, and the creation of that variable is the only reason to use it.

Common Lisp itself does not treat this convention as inviolable. The binding of the CLOS function `call-next-method` depends on the context in just the same way that the binding of the symbol `it` does within the body of an `aif`. (For a suggestion of how `call-next-method` would be implemented, see the macro `defmeth` on page 358.) In any case, such conventions are only supposed to be a means to an end: programs which are easy to read. And anaphora do make programs easier to read, just as they make English easier to read.

15

Macros Returning Functions

Chapter 5 showed how to write functions which return other functions. Macros make the task of combining operators much easier. This chapter will show how to use macros to build abstractions which are equivalent to those defined in Chapter 5, but cleaner and more efficient.

15.1 Building Functions

If f and g are functions, then $f \circ g(x) = f(g(x))$. Section 5.4 showed how to implement the $\circ$ operator as a Lisp function called `compose`:

```
> (funcall (compose #'list #'1+) 2)
(3)
```

In this section, we consider ways to define better function builders with macros. Figure 15.1 contains a general function-builder called `fn`, which builds compound functions from their descriptions. Its argument should be an expression of the form `(`*operator* `.` *arguments*`)`. The *operator* can be the name of a function or macro—or `compose`, which is treated specially. The *arguments* can be names of functions or macros of one argument, or expressions that could be arguments to `fn`. For example,

```
(fn (and integerp oddp))
```

yields a function equivalent to

```
#'(lambda (x) (and (integerp x) (oddp x)))
```

```
(defmacro fn (expr) `#',(rbuild expr))

(defun rbuild (expr)
  (if (or (atom expr) (eq (car expr) 'lambda))
      expr
      (if (eq (car expr) 'compose)
          (build-compose (cdr expr))
          (build-call (car expr) (cdr expr)))))

(defun build-call (op fns)
  (let ((g (gensym)))
    `(lambda (,g)
       (,op ,@(mapcar #'(lambda (f)
                          `(,(rbuild f) ,g))
                      fns)))))

(defun build-compose (fns)
  (let ((g (gensym)))
    `(lambda (,g)
       ,(labels ((rec (fns)
                   (if fns
                       `(,(rbuild (car fns))
                         ,(rec (cdr fns)))
                       g)))
          (rec fns)))))
```

Figure 15.1: General function-building macro.

If we use `compose` as the operator, we get a function representing the composition of the arguments, but without the explicit `funcall`s that were needed when compose was defined as a function. For example,

```
(fn (compose list 1+ truncate))
```

expands into:

```
#'(lambda (#:g1) (list (1+ (truncate #:g1))))
```

which enables inline compilation of simple functions like `list` and `1+`. The `fn` macro takes names of operators in the general sense; lambda-expressions are allowed too, as in

```
(fn (compose (lambda (x) (+ x 3)) truncate))
```

which expands into

```
#'(lambda (#:g2) ((lambda (x) (+ x 3)) (truncate #:g2)))
```

Here the function expressed as a lambda-expression will certainly be compiled inline, whereas a sharp-quoted lambda-expression given as an argument to the function `compose` would have to be funcalled.

Section 5.4 showed how to define three more function builders: `fif`, `fint`, and `fun`. These are now subsumed in the general `fn` macro. Using `and` as the operator yields the intersection of the operators given as arguments:

```
> (mapcar (fn (and integerp oddp))
          '(c 3 p 0))
(NIL T NIL NIL)
```

while `or` yields the union:

```
> (mapcar (fn (or integerp symbolp))
          '(c 3 p 0.2))
(T T T NIL)
```

and `if` yields a function whose body is a conditional:

```
> (map1-n (fn (if oddp 1+ identity)) 6)
(2 2 4 4 6 6)
```

However, we can use other Lisp functions besides these three:

```
> (mapcar (fn (list 1- identity 1+))
          '(1 2 3))
((0 1 2) (1 2 3) (2 3 4))
```

and the arguments in the `fn` expression may themselves be expressions:

```
> (remove-if (fn (or (and integerp oddp)
                     (and consp cdr)))
             '(1 (a b) c (d) 2 3.4 (e f g)))
(C (D) 2 3.4)
```

Making `fn` treat `compose` as a special case does not make it any more powerful. If you nest the arguments to `fn`, you get functional composition. For example,

```
(fn (list (1+ truncate)))
```

expands into:

```
#'(lambda (#:g1)
    (list ((lambda (#:g2) (1+ (truncate #:g2))) #:g1)))
```

which behaves like

```
(compose #'list #'1+ #'truncate)
```

The `fn` macro treats `compose` as a special case only to make such calls easier to read.

15.2 Recursion on Cdrs

Sections 5.5 and 5.6 showed how to write functions that build recursive functions. The following two sections show how anaphoric macros can provide a cleaner interface to the functions we defined there.

Section 5.5 showed how to define a flat list recurser builder called `lrec`. With `lrec` we can express a call to:

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

for e.g. `oddp` as:

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f)))
      t)
```

Here macros could make life easier. How much do we really have to say to express recursive functions? If we can refer anaphorically to the current car of the list (as `it`) and the recursive call (as `rec`), we should be able to make do with something like:

```
(alrec (and (oddp it) rec) t)
```

Figure 15.2 contains the definition of the macro which will allow us to say this.

```
> (funcall (alrec (and (oddp it) rec) t)
           '(1 3 5))
T
```

```
(defmacro alrec (rec &optional base)
  "cltl2 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (symbol-macrolet ((rec (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro alrec (rec &optional base)
  "cltl1 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (labels ((rec () (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro on-cdrs (rec base &rest lsts)
  `(funcall (alrec ,rec #'(lambda () ,base)) ,@lsts))
```

Figure 15.2: Macros for list recursion.

The new macro works by transforming the expression given as the second argument into a function to be passed to `lrec`. Since the second argument may refer anaphorically to `it` or `rec`, in the macro expansion the body of the function must appear within the scope of bindings established for these symbols.

Figure 15.2 actually has two different versions of `alrec`. The version used in the preceding examples requires symbol macros (Section 7.11). Only recent versions of Common Lisp have symbol macros, so Figure 15.2 also contains a slightly less convenient version of `alrec` in which `rec` is defined as a local function. The price is that, as a function, `rec` would have to be enclosed within parentheses:

```
(alrec (and (oddp it) (rec)) t)
```

The original version is preferable in Common Lisp implementations which provide `symbol-macrolet`.

Common Lisp, with its separate name-space for functions, makes it awkward to use these recursion builders to define named functions:

```
(setf (symbol-function 'our-length)
      (alrec (1+ rec) 0))
```

```
(defun our-copy-list (lst)
  (on-cdrs (cons it rec) nil lst))

(defun our-remove-duplicates (lst)
  (on-cdrs (adjoin it rec) nil lst))

(defun our-find-if (fn lst)
  (on-cdrs (if (funcall fn it) it rec) nil lst))

(defun our-some (fn lst)
  (on-cdrs (or (funcall fn it) rec) nil lst))
```

Figure 15.3: Common Lisp functions defined with `on-cdrs`.

The final macro in Figure 15.2 is intended to make this more abstract. Using `on-cdrs` we could say instead:

```
(defun our-length (lst)
  (on-cdrs (1+ rec) 0 lst))

(defun our-every (fn lst)
  (on-cdrs (and (funcall fn it) rec) t lst))
```

Figure 15.3 shows some existing Common Lisp functions defined with the new macro. Expressed with `on-cdrs`, these functions are reduced to their most basic form, and we notice similarities between them which might not otherwise have been apparent.

Figure 15.4 contains some new utilities which can easily be defined with `on-cdrs`. The first three, `unions`, `intersections`, and `differences` implement set union, intersection, and complement, respectively. Common Lisp has built-in functions for these operations, but they can only take two lists at a time. Thus if we want to find the union of three lists we have to say:

```
> (union '(a b) (union '(b c) '(c d)))
(A B C D)
```

The new `unions` behaves like `union`, but takes an arbitrary number of arguments, so that we could say:

```
> (unions '(a b) '(b c) '(c d))
(D C A B)
```

```
(defun unions (&rest sets)
  (on-cdrs (union it rec) (car sets) (cdr sets)))

(defun intersections (&rest sets)
  (unless (some #'null sets)
    (on-cdrs (intersection it rec) (car sets) (cdr sets))))

(defun differences (set &rest outs)
  (on-cdrs (set-difference rec it) set outs))

(defun maxmin (args)
  (when args
    (on-cdrs (multiple-value-bind (mx mn) rec
               (values (max mx it) (min mn it)))
             (values (car args) (car args))
             (cdr args))))
```

Figure 15.4: New utilities defined with `on-cdrs`.

Like `union`, `unions` does not preserve the order of the elements in the initial lists.

The same relation holds between the Common Lisp `intersection` and the more general `intersections`. In the definition of this function, the initial test for null arguments was added for efficiency; it short-circuits the computation if one of the sets is empty.

Common Lisp also has a function called `set-difference`, which takes two lists and returns the elements of the first which are not in the second:

```
> (set-difference '(a b c d) '(a c))
(D B)
```

Our new version handles multiple arguments much as `-` does. For example, `(differences x y z)` is equivalent to `(set-difference x (unions y z))`, though without the consing that the latter would entail.

```
> (differences '(a b c d e) '(a f) '(d))
(B C E)
```

These set operators are intended only as examples. There is no real need for them, because they represent a degenerate case of list recursion already handled by the built-in `reduce`. For example, instead of

```
(unions ...)
```

you might as well say just

```
((lambda (&rest args) (reduce #'union args)) ...)
```

In the general case, `on-cdrs` is more powerful than `reduce`, however.

Because `rec` refers to a call instead of a value, we can use `on-cdrs` to create functions which return multiple values. The final function in Figure 15.4, `maxmin`, takes advantage of this possibility to find both the maximum and minimum elements in a single traversal of a list:

```
> (maxmin '(3 4 2 8 5 1 6 7))
8
1
```

It would also have been possible to use `on-cdrs` in some of the code which appears in later chapters. For example, `compile-cmds` (page 310)

```
(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds)))))
```

could have been defined as simply:

```
(defun compile-cmds (cmds)
  (on-cdrs `(,@it ,rec) 'regs cmds))
```

15.3 Recursion on Subtrees

What macros did for recursion on lists, they can also do for recursion on trees. In this section, we use macros to define cleaner interfaces to the tree recursers defined in Section 5.6.

In Section 5.6 we defined two tree recursion builders, `ttrav`, which always traverses the whole tree, and `trec` which is more complex, but allows you to control when recursion stops. Using these functions we could express `our-copy-tree`

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

as

```
(ttrav #'cons)
```

and a call to `rfind-if`

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (and (cdr tree) (rfind-if fn (cdr tree))))))
```

for e.g. `oddp` as:

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

Anaphoric macros can make a better interface to `trec`, as they did for `lrec` in the previous section. A macro sufficient for the general case will have to be able to refer anaphorically to three things: the current tree, which we'll call `it`, the recursion down the left subtree, which we'll call `left`, and the recursion down the right subtree, which we'll call `right`. With these conventions established, we should be able to express the preceding functions in terms of a new macro thus:

```
(atrec (cons left right))

(atrec (or left right) (and (oddp it) it))
```

Figure 15.5 contains the definition of this macro.

In versions of Lisp which don't have `symbol-macrolet`, we can define `atrec` using the second definition in Figure 15.5. This version defines `left` and `right` as local functions, so `our-copy-tree` would have to be expressed as:

```
(atrec (cons (left) (right)))
```

For convenience, we also define a macro `on-trees`, which is analogous to `on-cdrs` from the previous section. Figure 15.6 shows the four functions from Section 5.6 defined with `on-trees`.

As noted in Chapter 5, functions built by the recurser generators defined in that chapter will not be tail-recursive. Using `on-cdrs` or `on-trees` to define a function will not necessarily yield the most efficient implementation. Like the underlying `trec` and `lrec`, these macros are mainly for use in prototypes and in parts of a program where efficiency is not paramount. However, the underlying idea of this chapter and Chapter 5 is that one can write function generators and put a clean macro interface on them. This same technique could equally well be used to build function generators which yielded particularly efficient code.

```
(defmacro atrec (rec &optional (base 'it))
  "cltl2 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (symbol-macrolet ((left (funcall ,lfn))
                                 (right (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro atrec (rec &optional (base 'it))
  "cltl1 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (labels ((left () (funcall ,lfn))
                        (right () (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro on-trees (rec base &rest trees)
  `(funcall (atrec ,rec ,base) ,@trees))
```

Figure 15.5: Macros for recursion on trees.

```
(defun our-copy-tree (tree)
  (on-trees (cons left right) it tree))

(defun count-leaves (tree)
  (on-trees (+ left (or right 1)) 1 tree))

(defun flatten (tree)
  (on-trees (nconc left right) (mklist it) tree))

(defun rfind-if (fn tree)
  (on-trees (or left right)
            (and (funcall fn it) it)
            tree))
```

Figure 15.6: Functions defined using `on-trees`.

```
(defconstant unforced (gensym))

(defstruct delay  forced closure)

(defmacro delay (expr)
  (let ((self (gensym)))
    `(let ((,self (make-delay :forced unforced)))
       (setf (delay-closure ,self)
             #'(lambda ()
                 (setf (delay-forced ,self) ,expr)))
       ,self)))

(defun force (x)
  (if (delay-p x)
      (if (eq (delay-forced x) unforced)
          (funcall (delay-closure x))
          (delay-forced x))
      x))
```

Figure 15.7: Implementation of `force` and `delay`.

15.4 Lazy Evaluation

Lazy evaluation means only evaluating an expression when you need its value. One way to use lazy evaluation is to build an object known as a *delay*. A delay is a placeholder for the value of some expression. It represents a promise to deliver the value of the expression if it is needed at some later time. Meanwhile, since the promise is a Lisp object, it can serve many of the purposes of the value it represents. And when the value of the expression is needed, the delay can return it.

Scheme has built-in support for delays. The Scheme operators `force` and `delay` can be implemented in Common Lisp as in Figure 15.7. A delay is represented as a two-part structure. The first field indicates whether the delay has been evaluated yet, and if it has, contains the value. The second field contains a closure which can be called to find the value that the delay represents. The macro `delay` takes an expression, and returns a delay representing its value:

```
> (let ((x 2))
    (setq d (delay (1+ x))))
#S(DELAY ...)
```

To call the closure within a delay is to *force* the delay. The function `force` takes any object: for ordinary objects it is the identity function, but for delays it is a demand for the value that the delay represents.

```
> (force 'a)
A
> (force d)
3
```

We use `force` whenever we are dealing with objects that might be delays. For example, if we are sorting a list which might contain delays, we would say:

```
(sort lst #'(lambda (x y) (> (force x) (force y))))
```

It's slightly inconvenient to use delays in this naked form. In a real application, they might be hidden beneath another layer of abstraction.

16

Macro-Defining Macros

Patterns in code often signal the need for new abstractions. This rule holds just as much for the code in macros themselves. When several macros have definitions of a similar form, we may be able to write a macro-defining macro to produce them. This chapter presents three examples of macro-defining macros: one to define abbreviations, one to define access macros, and a third to define anaphoric macros of the type described in Section 14.1.

16.1 Abbreviations

The simplest use of macros is as abbreviations. Some Common Lisp operators have rather long names. Ranking high among them (though by no means the longest) is `destructuring-bind`, which has 18 characters. A corollary of ◦
Steele's principle (page 43) is that commonly used operators ought to have short names. ("We think of addition as cheap partly because we can notate it with a single character: '+'.") The built-in `destructuring-bind` macro introduces a new layer of abstraction, but the actual gain in brevity is masked by its long name:

```
(let ((a (car x)) (b (cdr x))) ...)

(destructuring-bind (a . b) x ...)
```

A program, like printed text, is easiest to read when it contains no more than about 70 characters per line. We begin at a disadvantage when the lengths of individual names are a quarter of that.

```
(defmacro abbrev (short long)
  `(defmacro ,short (&rest args)
     `(,',long ,@args)))

(defmacro abbrevs (&rest names)
  `(progn
     ,@(mapcar #'(lambda (pair)
                   `(abbrev ,@pair))
               (group names 2))))
```

Figure 16.1: Automatic definition of abbreviations.

Fortunately, in a language like Lisp you don't have to live with all the decisions of the designers. Having defined

```
(defmacro dbind (&rest args)
  `(destructuring-bind ,@args))
```

you need never use the long name again. Likewise for `multiple-value-bind`, which is longer and more frequently used.

```
(defmacro mvbind (&rest args)
  `(multiple-value-bind ,@args))
```

Notice how nearly identical are the definitions of `dbind` and `mvbind`. Indeed, this formula of `&rest` and comma-at will suffice to define an abbreviation for any function,[1] macro, or special form. Why crank out more definitions on the model of `mvbind` when we could have a macro do it for us?

To define a macro-defining macro we will often need nested backquotes. Nested backquotes are notoriously hard to understand. Eventually common cases will become familiar, but one should not expect to be able to look at an arbitrary backquoted expression and say what it will yield. It is not a fault in Lisp that this is so, any more than it is a fault of the notation that one can't just look at a complicated integral and know what its value will be. The difficulty is in the problem, not the notation.

However, as we do when finding integrals, we can break up the analysis of backquotes into small steps, each of which can easily be followed. Suppose we want to write a macro `abbrev`, which will allow us to define `mvbind` just by saying

[1]Though the abbreviation can't be passed to `apply` or `funcall`.

```
(abbrev mvbind multiple-value-bind)
```

Figure 16.1 contains a definition of this macro. Where did it come from? The definition of such a macro can be derived from a sample expansion. One expansion is:

```
(defmacro mvbind (&rest args)
  `(multiple-value-bind ,@args))
```

The derivation will be easier if we pull `multiple-value-bind` from within the backquote, because we know it will be an argument to the eventual macro. This yields the equivalent definition

```
(defmacro mvbind (&rest args)
  (let ((name 'multiple-value-bind))
    `(,name ,@args)))
```

Now we take this expression and turn it into a template. We affix a backquote, and replace the expressions which will vary, with variables.

```
`(defmacro ,short (&rest args)
   (let ((name ',long))
     `(,name ,@args)))
```

The final step is to simplify this expression by substituting `',long` for `name` within the inner backquote:

```
`(defmacro ,short (&rest args)
   `(,',long ,@args))
```

which yields the body of the macro defined in Figure 16.1.

Figure 16.1 also contains `abbrevs`, for cases where we want to define several abbreviations in one shot.

```
(abbrevs dbind  destructuring-bind
         mvbind multiple-value-bind
         mvsetq multiple-value-setq)
```

The user of `abbrevs` doesn't have to insert additional parentheses because `abbrevs` calls `group` (page 47) to group its arguments by twos. It's generally a good thing for macros to save users from typing logically unnecessary parentheses, and `group` will be useful to most such macros.

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
     `(get ,obj ',',propname)))

(defmacro propmacros (&rest props)
  `(progn
     ,@(mapcar #'(lambda (p) `(propmacro ,p))
               props)))
```

Figure 16.2: Automatic definition of access macros.

16.2 Properties

Lisp offers many ways to associate properties with objects. If the object in question can be represented as a symbol, one of the most convenient (though least efficient) ways is to use the symbol's property list. To describe the fact that an object *o* has a property *p*, the value of which is *v*, we modify the property list of *o*:

```
(setf (get o p) v)
```

So to say that `ball1` has `color red`, we say:

```
(setf (get 'ball1 'color) 'red)
```

If we're going to refer often to some property of objects, we can define a macro to retrieve it:

```
(defmacro color (obj)
  `(get ,obj 'color))
```

and thereafter use `color` in place of `get`:

```
> (color 'ball1)
RED
```

Since macro calls are transparent to `setf` (see Chapter 12) we can also say:

```
> (setf (color 'ball1) 'green)
GREEN
```

Such macros have the advantage of hiding the particular way in which the program represents the color of an object. Property lists are slow; a later version of the program might, for the sake of speed, represent color as a field in a structure, or an entry in a hash-table. When data is reached through a facade of macros like `color`, it becomes easy, even in a comparatively mature program, to make pervasive changes to the lowest-level code. If a program switches from using property lists to structures, nothing above the facade of access macros will have to be changed; none of the code which looks upon the facade need even be aware of the rebuilding going on behind it.

For the weight property, we can define a macro similar to the one written for color:

```
(defmacro weight (obj)
  `(get ,obj 'weight))
```

Like the abbreviations in the previous section, the definitions of of `color` and `weight` are nearly identical. Here `propmacro` (Figure 16.2) can play the same role as `abbrev` did.

A macro-defining macro can be designed by the same process as any other macro: look at the macro call, then its intended expansion, then figure out how to transform the former into the latter. We want

```
(propmacro color)
```

to expand into

```
(defmacro color (obj)
  `(get ,obj 'color))
```

Though this expression is itself a `defmacro`, we can still make a template of it, by backquoting it and putting comma'd parameter names in place of instances of `color`. As in the previous section, we begin by transforming it so that no instances of `color` are within existing backquotes:

```
(defmacro color (obj)
  (let ((p 'color))
    `(get ,obj ',p)))
```

Then we go ahead and make the template,

```
`(defmacro ,propname (obj)
   (let ((p ',propname))
     `(get ,obj ',p)))
```

which simplifies to ◦

```
‘(defmacro ,propname (obj)
   ‘(get ,obj ’,’,propname))
```

For cases where a group of property-names all have to be defined as macros, there is propmacros (Figure 16.2), which expands into a series of individual calls to propmacro. Like abbrevs, this modest piece of code is actually a macro-defining-macro-defining macro.

Though this section dealt with property lists, the technique described here is a general one. We could use it to define access macros on data stored in any form.

16.3 Anaphoric Macros

Section 14.1 gave definitions of several anaphoric macros. When you use a macro like aif or aand, during the evaluation of some arguments the symbol it will be bound to the values returned by other ones. So instead of

```
(let ((res (complicated-query)))
  (if res
      (foo res)))
```

you can use just

```
(aif (complicated-query)
     (foo it))
```

and instead of

```
(let ((o (owner x)))
  (and o (let ((a (address o)))
           (and a (city a)))))
```

simply

```
(aand (owner x) (address it) (city it))
```

Section 14.1 presented seven anaphoric macros: aif, awhen, awhile, acond, alambda, ablock, and aand. These seven are by no means the only useful anaphoric macros of their type. In fact, we can define an anaphoric variant of just about any Common Lisp function or macro. Many of these macros will be like mapcon: rarely used, but indispensable when they are needed.

For example, we can define a+ so that, as with aand, it is always bound to the value returned by the previous argument. The following function calculates the cost of dining out in Massachusetts:

```
(defmacro a+ (&rest args)
  (a+expand args nil))

(defun a+expand (args syms)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(a+expand (cdr args)
                      (append syms (list sym)))))
      `(+ ,@syms)))

(defmacro alist (&rest args)
  (alist-expand args nil))

(defun alist-expand (args syms)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(alist-expand (cdr args)
                          (append syms (list sym)))))
      `(list ,@syms)))
```

Figure 16.3: Definitions of `a+` and `alist`.

```
(defun mass-cost (menu-price)
  (a+ menu-price (* it .05) (* it 3)))
```

The Massachusetts Meals Tax is 5%, and residents often calculate the tip by tripling the tax. By this formula, the total cost of the broiled scrod at Dolphin Seafood is therefore:

```
> (mass-cost 7.95)
9.54
```

but this includes salad and a baked potato.

The macro `a+`, defined in Figure 16.3, relies on a recursive function, `a+expand`, to generate its expansion. The general strategy of `a+expand` is to cdr down the list of arguments in the macro call, generating a series of nested `let` expressions; each `let` leaves `it` bound to a different argument, but also binds a distinct gensym

```
(defmacro defanaph (name &optional calls)
   (let ((calls (or calls (pop-symbol name))))
    `(defmacro ,name (&rest args)
       (anaphex args (list ',calls)))))

(defun anaphex (args expr)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(anaphex (cdr args)
                     (append expr (list sym)))))
      expr))

(defun pop-symbol (sym)
  (intern (subseq (symbol-name sym) 1)))
```

Figure 16.4: Automatic definition of anaphoric macros.

to each argument. The expansion function accumulates a list of these gensyms, and when it reaches the end of the list of arguments it returns a + expression with the gensyms as the arguments. So the expression

```
(a+ menu-price (* it .05) (* it 3))
```

yields the macroexpansion:

```
(let* ((#:g2 menu-price) (it #:g2))
  (let* ((#:g3 (* it 0.05)) (it #:g3))
    (let* ((#:g4 (* it 3)) (it #:g4))
      (+ #:g2 #:g3 #:g4))))
```

Figure 16.3 also contains the definition of the analogous `alist`:

```
> (alist 1 (+ 2 it) (+ 2 it))
(1 3 5)
```

Once again, the definitions of `a+` and `alist` are almost identical. If we want to define more macros like them, these too will be mostly duplicate code. Why not have a program produce it for us? The macro `defanaph` in Figure 16.4 will do so. With `defanaph`, defining `a+` and `alist` is as simple as

```
(defanaph a+)
(defanaph alist)
```

The expansions of a+ and alist so defined will be identical to the expansions made by the code in Figure 16.3. The macro-defining macro defanaph will create an anaphoric variant of anything whose arguments are evaluated according to the normal evaluation rule for functions. That is, defanaph will work for anything whose arguments are all evaluated, and evaluated left-to-right. So you couldn't use this version of defanaph to define aif or awhile, but you can use it to define an anaphoric variant of any function.

As a+ called a+expand to generate its expansion, defanaph defines a macro which will call anaphex to do so. The generic expander anaphex differs from a+expand only in taking as an argument the function name to appear finally in the expansion. In fact, a+ could now be defined:

```
(defmacro a+ (&rest args)
  (anaphex args '(+)))
```

Neither anaphex nor a+expand need have been defined as distinct functions: anaphex could have been defined with labels or alambda within defanaph. The expansion generators are here broken out as separate functions only for the sake of clarity.

By default, defanaph determines what to call in the expansion by pulling the first letter (presumably an a) from the front of its argument. (This operation is performed by pop-symbol.) If the user prefers to specify an alternate name, it can be given as an optional argument. Although defanaph can build anaphoric variants of all functions and some macros, it imposes some irksome restrictions:

1. It only works for operators whose arguments are all evaluated.
2. In the macroexpansion, it is always bound to successive arguments. In some cases—awhen, for example—we want it to stay bound to the value of the *first* argument.
3. It won't work for a macro like setf, which expects a generalized variable as its first argument.

Let's consider how to remove some of these restrictions. Part of the first problem can be solved by solving the second. To generate expansions for a macro like aif, we need a modified version of anaphex which only replaces the first argument in the macro call:

```
(defun anaphex2 (op args)
  `(let ((it ,(car args)))
     (,op it ,@(cdr args))))
```

This nonrecursive version of anaphex doesn't need to ensure that the macroexpansion will bind it to successive arguments, so it can generate an expansion which won't necessarily evaluate all the arguments in the macro call. Only the first argument must be evaluated, in order to bind it to its value. So aif could be defined as:

```
(defmacro aif (&rest args)
  (anaphex2 'if args))
```

This definition would differ from the original on page 191 only in the point where it would complain if aif were given the wrong number of arguments; for correct macro calls, the two generate identical expansions.

The third problem, that defanaph won't work with generalized variables, can be solved by using _f (page 173) in the expansion. Operators like setf can be handled by a variant of anaphex2 defined as follows:

```
(defun anaphex3 (op args)
  `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

This expander assumes that the macro call will have one or more arguments, the first of which will be a generalized variable. Using it we could define asetf thus:

```
(defmacro asetf (&rest args)
  (anaphex3 'setf args))
```

Figure 16.5 shows all three expander functions yoked together under the control of a single macro, the new defanaph. The user signals the type of macro expansion desired with the optional rule keyword parameter, which specifies the evaluation rule to be used for the arguments in the macro call. If this parameter is:

:all (the default) the macroexpansion will be on the model of alist. All the arguments in the macro call will be evaluated, with it always bound to the value of the previous argument.

:first the macroexpansion will be on the model of aif. Only the first argument will necessarily be evaluated, and it will be bound to its value.

:place the macroexpansion will be on the model of asetf. The first argument will be treated as a generalized variable, and it will be bound to its initial value.

Using the new defanaph, some of the previous examples would be defined as follows:

```
(defmacro defanaph (name &optional &key calls (rule :all))
  (let* ((opname (or calls (pop-symbol name)))
         (body (case rule
                 (:all   `(anaphex1 args '(,opname)))
                 (:first `(anaphex2 ',opname args))
                 (:place `(anaphex3 ',opname args)))))
    `(defmacro ,name (&rest args)
       ,body)))

(defun anaphex1 (args call)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(anaphex1 (cdr args)
                      (append call (list sym)))))
      call))

(defun anaphex2 (op args)
  `(let ((it ,(car args))) (,op it ,@(cdr args))))

(defun anaphex3 (op args)
  `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

Figure 16.5: More general defanaph.

```
(defanaph alist)
(defanaph aif :rule :first)
(defanaph asetf :rule :place)
```

One of the advantages of asetf is that it makes it possible to define a large class of macros on generalized variables without worrying about multiple evaluation. For example, we could define incf as:

```
(defmacro incf (place &optional (val 1))
  `(asetf ,place (+ it ,val)))
```

and, say, pull (page 173) as:

```
(defmacro pull (obj place &rest args)
  `(asetf ,place (delete ,obj it ,@args)))
```

17

Read-Macros

The three big moments in a Lisp expression's life are read-time, compile-time, and runtime. Functions are in control at runtime. Macros give us a chance to perform transformations on programs at compile-time. This chapter discusses read-macros, which do their work at read-time.

17.1 Macro Characters

In keeping with the general philosophy of Lisp, you have a great deal of control over the reader. Its behavior is controlled by properties and variables that can all be changed on the fly. The reader can be programmed at several levels. The easiest way to change its behavior is by defining new macro characters.

A *macro character* is a character which exacts special treatment from the Lisp reader. A lower-case `a`, for example, is ordinarily handled just like a lower-case `b`, but a left parenthesis is something different: it tells Lisp to begin reading a list. Each such character has a function associated with it that tells the Lisp reader what to do when the character is encountered. You can change the function associated with an existing macro character, or define new macro characters of your own.

The built-in function `set-macro-character` provides one way to define read-macros. It takes a character and a function, and thereafter when `read` encounters the character, it returns the result of calling the function.

One of the oldest read-macros in Lisp is `'`, the quote. You could do without `'` by always writing `(quote a)` instead of `'a`, but this would be tiresome and would make your code harder to read. The quote read-macro makes it possible to use `'a` as an abbreviation for `(quote a)`. We could define it as in Figure 17.1.

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list 'quote (read stream t nil t))))
```

Figure 17.1: Possible definition of `'`.

When `read` encounters an instance of `'` in a normal context (e.g. not in `"a'b"` or `|a'b|`), it will return the result of calling this function on the current stream and character. (The function ignores this second parameter, which will always be the quote character.) So when `read` sees `'a`, it will return `(quote a)`.

The last three arguments to `read` control respectively whether encountering an end-of-file should cause an error, what value to return otherwise, and whether the call to `read` occurs within a call to `read`. In nearly all read-macros, the second and fourth arguments should be `t`, and the third argument is therefore irrelevant.

Read-macros and ordinary macros are both functions underneath. And like the functions that generate macro expansions, the functions associated with macro characters shouldn't have side-effects, except on the stream from which they read. Common Lisp explicitly makes no guarantees about when, or how often, the function associated with a read-macro will be called. (See CLTL2, p. 543.)

Macros and read-macros see your program at different stages. Macros get hold of the program when it has already been parsed into Lisp objects by the reader, and read-macros operate on a program while it is still text. However, by invoking `read` on this text, a read-macro can, if it chooses, get parsed Lisp objects as well. Thus read-macros are at least as powerful as ordinary macros.

Indeed, read-macros are more powerful in at least two ways. A read-macro affects everything read by Lisp, while a macro will only be expanded in code. And since read-macros generally call `read` recursively, an expression like

```
''a
```

becomes

```
(quote (quote a))
```

whereas if we had tried to define an abbreviation for `quote` using a normal macro,

```
(defmacro q (obj)
  `(quote ,obj))
```

```
(set-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      `#'(lambda (&rest ,(gensym))
           ,(read stream t nil t))))
```

Figure 17.2: A read-macro for constant functions.

it would work in isolation,

```
> (eq 'a (q a))
T
```

but not when nested. For example,

```
(q (q a))
```

would expand into

```
(quote (q a))
```

17.2 Dispatching Macro Characters

The sharp-quote, like other read-macros beginning with `#`, is an example of a subspecies called *dispatching* read-macros. These appear as two characters, the first of which is called the dispatching character. The purpose of such read-macros is simply to make the most of the ASCII character set; one can only have so many one-character read-macros.

You can (with `make-dispatch-macro-character`) define your own dispatching macro characters, but since `#` is already defined as one, you may as well use it. Some combinations beginning with `#` are explicitly reserved for your use; others are available in that they do not yet have a predefined meaning in Common Lisp. The complete list appears in CLTL2, p. 531.

New dispatching macro character combinations can be defined by calling the function `set-dispatch-macro-character`, like `set-macro-character` except that it takes two character arguments. One of the combinations reserved to the programmer is `#?`. Figure 17.2 shows how to define this combination as a read-macro for constant functions. Now `#?2` will be read as a function which takes any number of arguments and returns 2. For example:

```
> (mapcar #?2 '(a b c))
(2 2 2)
```

```
(set-macro-character #\] (get-macro-character #\)))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\] stream t)))
        (do ((i (ceiling (car pair)) (1+ i)))
            ((> i (floor (cadr pair)))
             (list 'quote (nreverse accum)))
          (push i accum)))))
```

Figure 17.3: A read-macro defining delimiters.

This example makes the new operator look rather pointless, but in programs that use a lot of functional arguments, constant functions are often needed. In fact, some dialects provide a built-in function called `always` for defining them.

Note that it is perfectly ok to use macro characters in the definition of this macro character: as with any Lisp expression, they disappear when the definition is read. It is also fine to use macro-characters after the `#?`. The definition of `#?` calls `read`, so macro-characters like `'` and `#'` behave as usual:

```
> (eq (funcall #?'a) 'a)
T
> (eq (funcall #?#'oddp) (symbol-function 'oddp))
T
```

17.3 Delimiters

After simple macro characters, the most commonly defined macro characters are list delimiters. Another character combination reserved for the user is `#[`. Figure 17.3 gives an example of how this character might be defined as a more elaborate kind of left parenthesis. It defines an expression of the form `#[x y]` to read as a list of all the integers between `x` and `y`, inclusive:

```
> #[2 7]
(2 3 4 5 6 7)
```

The only new thing about this read-macro is the call to `read-delimited-list`, a built-in function provided just for such cases. Its first argument is the character to treat as the end of the list. For `]` to be recognized as a delimiter, it must first be given this role, hence the preliminary call to `set-macro-character`.

```
(defmacro defdelim (left right parms &body body)
  `(ddfn ,left ,right #'(lambda ,parms ,@body)))

(let ((rpar (get-macro-character #\) )))
  (defun ddfn (left right fn)
    (set-macro-character right rpar)
    (set-dispatch-macro-character #\# left
      #'(lambda (stream char1 char2)
          (apply fn
                 (read-delimited-list right stream t))))))
```

Figure 17.4: A macro for defining delimiter read-macros.

Most potential delimiter read-macro definitions will duplicate a lot of the code in Figure 17.3. A macro could put a more abstract interface on all this machinery. Figure 17.4 shows how we might define a utility for defining delimiter read-macros. The `defdelim` macro takes two characters, a parameter list, and a body of code. The parameter list and the body of code implicitly define a function. A call to `defdelim` defines the first character as a dispatching read-macro which reads up to the second, then returns the result of applying this function to what it read.

Incidentally, the body of the function in Figure 17.3 also cries out for a utility—for one we have already defined, in fact: `mapa-b`, from page 54. Using `defdelim` and `mapa-b`, the read-macro defined in Figure 17.3 could now be written:

```
(defdelim #\[ #\] (x y)
  (list 'quote (mapa-b #'identity (ceiling x) (floor y))))
```

Another useful delimiter read-macro would be one for functional composition. Section 5.4 defined an operator for functional composition:

```
> (let ((f1 (compose #'list #'1+))
        (f2 #'(lambda (x) (list (1+ x)))))
    (equal (funcall f1 7) (funcall f2 7)))
T
```

When we are composing built-in functions like `list` and `1+`, there is no reason to wait until runtime to evaluate the call to `compose`. Section 5.7 suggested an alternative; by prefixing the sharp-dot read-macro to a `compose` expression,

```
#.(compose #'list #'1+)
```

```
(defdelim #\{ #\} (&rest args)
  `(fn (compose ,@args)))
```

Figure 17.5: A read-macro for functional composition.

we could cause it to be evaluated at read-time.

Here we show a similar but cleaner solution. The read-macro in Figure 17.5 defines an expression of the form #{f_1 f_2 ... f_n} to read as the composition of $f_1, f_2, \ldots, f_n$. Thus:

```
> (funcall #{list 1+} 7)
(8)
```

It works by generating a call to `fn` (page 202), which will create the function at compile-time.

17.4 When What Happens

Finally, it might be useful to clear up a possibly confusing issue. If read-macros are invoked before ordinary macros, how is it that macros can expand into expressions which contain read-macros? For example, the macro:

```
(defmacro quotable ()
  '(list 'able))
```

generates an expansion with a quote in it. Or does it? In fact, what happens is that both quotes in the definition of this macro are expanded when the `defmacro` expression is read, yielding

```
(defmacro quotable ()
  (quote (list (quote able))))
```

Usually, there is no harm in acting as if macroexpansions could contain read-macros, because the definition of a read-macro will not (or should not) change between read-time and compile-time.

18

Destructuring

Destructuring is a generalization of assignment. The operators `setq` and `setf` do assignments to individual variables. Destructuring combines assignment with access: instead of giving a single variable as the first argument, we give a pattern of variables, which are each assigned the value occurring in the corresponding position in some structure.

18.1 Destructuring on Lists

As of CLTL2, Common Lisp includes a new macro called `destructuring-bind`. This macro was briefly introduced in Chapter 7. Here we consider it in more detail. Suppose that `lst` is a list of three elements, and we want to bind `x` to the first, `y` to the second, and `z` to the third. In raw CLTL1 Common Lisp, we would have had to say:

```
(let ((x (first lst))
      (y (second lst))
      (z (third lst)))
  ...)
```

With the new macro we can say instead

```
(destructuring-bind (x y z) lst
  ...)
```

which is not only shorter, but clearer as well. Readers grasp visual cues much faster than textual ones. In the latter form we are shown the relationship between x, y, and z; in the former, we have to infer it.

If such a simple case is made clearer by the use of destructuring, imagine the improvement in more complex ones. The first argument to destructuring-bind can be an arbitrarily complex tree. Imagine

```
(destructuring-bind ((first last) (month day year) . notes)
                    birthday
  ...)
```

written using let and the list access functions. Which raises another point: destructuring makes it easier to write programs as well as easier to read them.

Destructuring did exist in CLTL1 Common Lisp. If the patterns in the examples above look familiar, it's because they have the same form as macro parameter lists. In fact, destructuring-bind *is* the code used to take apart macro argument lists, now sold separately. You can put anything in the pattern that you would put in a macro parameter list, with one unimportant exception (the &environment keyword).

Establishing bindings *en masse* is an attractive idea. The following sections describe several variations upon this theme.

18.2 Other Structures

There is no reason to limit destructuring to lists. Any complex object is a candidate for it. This section shows how to write macros like destructuring-bind for other kinds of objects.

The natural next step is to handle sequences generally. Figure 18.1 contains a macro called dbind, which resembles destructuring-bind, but works for any kind of sequence. The second argument can be a list, a vector, or any combination thereof:

```
> (dbind (a b c) #(1 2 3)
    (list a b c))
(1 2 3)
> (dbind (a (b c) d) '( 1 #(2 3) 4)
    (list a b c d))
(1 2 3 4)
> (dbind (a (b . c) &rest d) '(1 "fribble" 2 3 4)
    (list a b c d))
(1 #\f "ribble" (2 3 4))
```

```
(defmacro dbind (pat seq &body body)
  (let ((gseq (gensym)))
    `(let ((,gseq ,seq))
       ,(dbind-ex (destruc pat gseq #'atom) body))))

(defun destruc (pat seq &optional (atom? #'atom) (n 0))
  (if (null pat)
      nil
      (let ((rest (cond ((funcall atom? pat) pat)
                        ((eq (car pat) '&rest) (cadr pat))
                        ((eq (car pat) '&body) (cadr pat))
                        (t nil))))
        (if rest
            `((,rest (subseq ,seq ,n)))
            (let ((p (car pat))
                  (rec (destruc (cdr pat) seq atom? (1+ n))))
              (if (funcall atom? p)
                  (cons `(,p (elt ,seq ,n))
                        rec)
                  (let ((var (gensym)))
                    (cons (cons `(,var (elt ,seq ,n))
                                (destruc p var atom?))
                          rec))))))))

(defun dbind-ex (binds body)
  (if (null binds)
      `(progn ,@body)
      `(let ,(mapcar #'(lambda (b)
                         (if (consp (car b))
                             (car b)
                             b))
                     binds)
         ,(dbind-ex (mapcan #'(lambda (b)
                                (if (consp (car b))
                                    (cdr b)))
                            binds)
                    body))))
```

Figure 18.1: General sequence destructuring operator.

The `#(` read-macro is for representing vectors, and `#\` for representing characters. Since `"abc"` = `#(#\a #\b #\c)`, the first element of `"fribble"` is the character `#\f`. For the sake of simplicity, `dbind` supports only the `&rest` and `&body` keywords.

Compared to most of the macros seen so far, `dbind` is big. It's worth studying the implementation of this macro, not only to understand how it works, but also because it embodies a general lesson about Lisp programming. As section 3.4 mentioned, Lisp programs may intentionally be written in a way that will make them easy to test. In most code, we have to balance this desire against the need for speed. Fortunately, as Section 7.8 explained, speed is not so important in expander code. When writing code that generates macroexpansions, we can make life easier for ourselves. The expansion of `dbind` is generated by two functions, `destruc` and `dbind-ex`. Perhaps they both could be combined into one function which would do everything in a single pass. But why bother? As two separate functions, they will be easier to test. Why trade this advantage for speed we don't need?

The first function, `destruc`, traverses the pattern and associates each variable with the location of the corresponding object at runtime:

```
> (destruc '(a b c) 'seq #'atom)
((A (ELT SEQ 0)) (B (ELT SEQ 1)) (C (ELT SEQ 2)))
```

The optional third argument is the predicate used to distinguish pattern structure from pattern content.

To make access more efficient, a new variable (a gensym) will be bound to each subsequence:

```
> (destruc '(a (b . c) &rest d) 'seq)
((A (ELT SEQ 0))
 ((#:G2 (ELT SEQ 1)) (B (ELT #:G2 0)) (C (SUBSEQ #:G2 1)))
 (D (SUBSEQ SEQ 2)))
```

The output of `destruc` is sent to `dbind-ex`, which generates the bulk of the macroexpansion. It translates the tree produced by `destruc` into a nested series of `let`s:

```
> (dbind-ex (destruc '(a (b . c) &rest d) 'seq) '(body))
(LET ((A (ELT SEQ 0))
      (#:G4 (ELT SEQ 1))
      (D (SUBSEQ SEQ 2)))
  (LET ((B (ELT #:G4 0))
        (C (SUBSEQ #:G4 1)))
    (PROGN BODY)))
```

```
(defmacro with-matrix (pats ar &body body)
  (let ((gar (gensym)))
    `(let ((,gar ,ar))
       (let ,(let ((row -1))
               (mapcan
                 #'(lambda (pat)
                     (incf row)
                     (setq col -1)
                     (mapcar #'(lambda (p)
                                 `(,p (aref ,gar
                                            ,row
                                            ,(incf col))))
                             pat))
                 pats))
         ,@body))))

(defmacro with-array (pat ar &body body)
  (let ((gar (gensym)))
    `(let ((,gar ,ar))
       (let ,(mapcar #'(lambda (p)
                         `(,(car p) (aref ,gar ,@(cdr p))))
                     pat)
         ,@body))))
```

Figure 18.2: Destructuring on arrays.

Note that `dbind`, like `destructuring-bind`, assumes that it will find all the list structure it is looking for. Left-over variables are not simply bound to `nil`, as with `multiple-value-bind`. If the sequence given at runtime does not have all the expected elements, destructuring operators generate an error:

```
> (dbind (a b c) (list 1 2))
>>Error: 2 is not a valid index for the sequence (1 2)
```

What other objects have internal structure? There are arrays generally, which differ from vectors in having more than one dimension. If we define a destructuring macro for arrays, how do we represent the pattern? For two-dimensional arrays, it is still practical to use a list. Figure 18.2 contains a macro, `with-matrix`, for destructuring on two-dimensional arrays.

```
(defmacro with-struct ((name . fields) struct &body body)
  (let ((gs (gensym)))
    `(let ((,gs ,struct))
       (let ,(mapcar #'(lambda (f)
                         `(,f (,(symb name f) ,gs)))
                     fields)
         ,@body))))
```

Figure 18.3: Destructuring on structures.

```
> (setq ar (make-array '(3 3)))
#<Simple-Array T (3 3) C2D39E>
> (for (r 0 2)
    (for (c 0 2)
      (setf (aref ar r c) (+ (* r 10) c))))
NIL
> (with-matrix ((a b c)
                (d e f)
                (g h i)) ar
    (list a b c d e f g h i))
(0 1 2 10 11 12 20 21 22)
```

For large arrays or those with dimension 3 or higher, we want a different kind of approach. We are not likely to want to bind variables to each element of a large array. It will be more practical to make the pattern a sparse representation of the array—containing variables for only a few elements, plus coordinates to identify them. The second macro in Figure 18.2 is built on this principle. Here we use it to get the diagonal of our previous array:

```
> (with-array ((a 0 0) (d 1 1) (i 2 2)) ar
    (values a d i))
0
11
22
```

With this new macro we have begun to move away from patterns whose elements must occur in a fixed order. We can make a similar sort of macro to bind variables to fields in structures built by `defstruct`. Such a macro is defined in Figure 18.3. The first argument in the pattern is taken to be the prefix associated with the structure, and the rest are field names. To build access calls, this macro uses `symb` (page 58).

```
> (defstruct visitor  name title firm)
VISITOR
> (setq theo (make-visitor :name  "Theodebert"
                           :title 'king
                           :firm  'franks))
#S(VISITOR NAME "Theodebert" TITLE KING FIRM FRANKS)
> (with-struct (visitor- name firm title) theo
    (list name firm title))
("Theodebert" FRANKS KING)
```

18.3 Reference

CLOS brings with it a macro for destructuring on instances. Suppose `tree` is a class with three slots, `species`, `age`, and `height`, and that `my-tree` is an instance of `tree`. Within

```
(with-slots (species age height) my-tree
  ...)
```

we can refer to the slots of `my-tree` as if they were ordinary variables. Within the body of the `with-slots`, the symbol `height` refers to the `height` slot. It is not simply bound to the value stored there, but *refers* to the slot, so that if we write:

```
(setq height 72)
```

then the `height` slot of `my-tree` will be given the value `72`. This macro works by defining `height` as a symbol-macro (Section 7.11) which expands into a slot reference. In fact, it was to support macros like `with-slots` that `symbol-macrolet` was added to Common Lisp.

Whether or not `with-slots` is really a destructuring macro, it has the same role pragmatically as `destructuring-bind`. As conventional destructuring is to call-by-value, this new kind is to call-by-name. Whatever we call it, it looks to be useful. What other macros can we define on the same principle?

We can create a call-by-name version of any destructuring macro by making it expand into a `symbol-macrolet` rather than a `let`. Figure 18.4 shows a version of `dbind` modified to behave like `with-slots`. We can use `with-places` as we do `dbind`:

```
> (with-places (a b c) #(1 2 3)
    (list a b c))
(1 2 3)
```

```
(defmacro with-places (pat seq &body body)
  (let ((gseq (gensym)))
    `(let ((,gseq ,seq))
       ,(wplac-ex (destruc pat gseq #'atom) body))))

(defun wplac-ex (binds body)
  (if (null binds)
      `(progn ,@body)
      `(symbol-macrolet ,(mapcar #'(lambda (b)
                                     (if (consp (car b))
                                         (car b)
                                         b))
                                 binds)
         ,(wplac-ex (mapcan #'(lambda (b)
                                (if (consp (car b))
                                    (cdr b)))
                            binds)
                    body))))
```

Figure 18.4: Reference destructuring on sequences.

But the new macro also gives us the option to `setf` positions in sequences, as we do slots in `with-slots`:

```
> (let ((lst '(1 (2 3) 4)))
    (with-places (a (b . c) d) lst
      (setf a 'uno)
      (setf c '(tre)))
    lst)
(UNO (2 TRE) 4)
```

As in a `with-slots`, the variables now refer to the corresponding locations in the structure. There is one important difference, however: you must use `setf` rather than `setq` to set these pseudo-variables. The `with-slots` macro must invoke a code-walker (page 273) to transform `setq`s into `setf`s within its body. Here, writing a code-walker would be a lot of code for a small refinement.

If `with-places` is more general than `dbind`, why not just use it all the time? While `dbind` associates a variable with a value, `with-places` associates it with a set of instructions for finding a value. Every reference requires a lookup. Where `dbind` would bind `c` to the value of `(elt x 2)`, `with-places` will make `c` a symbol-macro that expands into `(elt x 2)`. So if `c` is evaluated *n* times in the body, that will entail *n* calls to `elt`. Unless you actually want to `setf` the variables created by destructuring, `dbind` will be faster.

The definition of `with-places` is only slightly changed from that of `dbind` (Figure 18.1). Within `wplac-ex` (formerly `dbind-ex`) the `let` has become a `symbol-macrolet`. By similar alterations, we could make a call-by-name version of any normal destructuring macro.

18.4 Matching

As destructuring is a generalization of assignment, *pattern-matching* is a generalization of destructuring. The term "pattern-matching" has many senses. In this context, it means comparing two structures, possibly containing variables, to see if there is some way of assigning values to the variables which makes the two equal. For example, if `?x` and `?y` are variables, then the two lists

```
(p ?x ?y c ?x)
(p  a  b c  a)
```

match when `?x` = `a` and `?y` = `b`. And the lists

```
(p ?x b ?y a)
(p ?y b  c a)
```

match when `?x` = `?y` = `c`.

Suppose a program works by exchanging messages with some outside source. To respond to a message, the program has to tell what kind of message it is, and also to extract its specific content. With a matching operator we can combine the two steps.

To be able to write such an operator we have to invent some way of distinguishing variables. We can't just say that all symbols are variables, because we will want symbols to occur as arguments within patterns. Here we will say that a pattern variable is a symbol beginning with a question mark. If it becomes inconvenient, this convention could be changed simply by redefining the predicate `var?`.

Figure 18.5 contains a pattern-matching function similar to ones that appear ○ in several introductions to Lisp. We give `match` two lists, and if they can be made to match, we will get back a list showing how:

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
```

```
(defun match (x y &optional binds)
  (acond2
    ((or (eql x y) (eql x '_) (eql y '_)) (values binds t))
    ((binding x binds) (match it y binds))
    ((binding y binds) (match x it binds))
    ((varsym? x) (values (cons (cons x y) binds) t))
    ((varsym? y) (values (cons (cons y x) binds) t))
    ((and (consp x) (consp y) (match (car x) (car y) binds))
     (match (cdr x) (cdr y) it))
    (t (values nil nil))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (labels ((recbind (x binds)
             (aif (assoc x binds)
                  (or (recbind (cdr it) binds)
                      it))))
    (let ((b (recbind x binds)))
      (values (cdr b) b))))
```

Figure 18.5: Matching function.

```
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T
> (match '(a b c) '(a a a))
NIL
NIL
```

As `match` compares its arguments element by element, it builds up assignments of values to variables, called *bindings*, in the parameter `binds`. If the match is successful, `match` returns the bindings generated, otherwise it returns `nil`. Since not all successful matches generate any bindings, `match`, like `gethash`, returns a second value to indicate whether the match succeeded or failed:

```
> (match '(p ?x) '(p ?x))
NIL
T
```

```
(defmacro if-match (pat seq then &optional else)
  `(aif2 (match ',pat ,seq)
         (let ,(mapcar #'(lambda (v)
                           `(,v (binding ',v it)))
                       (vars-in then #'atom))
           ,then)
         ,else))

(defun vars-in (expr &optional (atom? #'atom))
  (if (funcall atom? expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr) atom?)
             (vars-in (cdr expr) atom?))))

(defun var? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

Figure 18.6: Slow matching operator.

When `match` returns `nil` and `t` as above, it indicates a successful match which yielded no bindings.

Like Prolog, `match` treats _ (underscore) as a wild-card. It matches everything, and has no effect on the bindings:

```
> (match '(a ?x b) '(_ 1 _))
((?X . 1))
T
```

Given `match`, it is easy to write a pattern-matching version of `dbind`. Figure 18.6 contains a macro called `if-match`. Like `dbind`, its first two arguments are a pattern and a sequence, and it establishes bindings by comparing the pattern with the sequence. However, instead of a body it has two more arguments: a `then` clause to be evaluated, with new bindings, if the match succeeds; and an `else` clause to be evaluated if the match fails. Here is a simple function which uses `if-match`:

```
(defun abab (seq)
  (if-match (?x ?y ?x ?y) seq
      (values ?x ?y)
      nil))
```

If the match succeeds, it will establish values for `?x` and `?y`, which will be returned:

```
> (abab '(hi ho hi ho))
HI
HO
```

The function `vars-in` returns all the pattern variables in an expression. It calls `var?` to test if something is a variable. At the moment, `var?` is identical to `varsym?` (Figure 18.5), which is used to detect variables in binding lists. We have two distinct functions in case we want to use different representations for the two kinds of variables.

As defined in Figure 18.6, `if-match` is short, but not very efficient. It does too much work at runtime. We traverse both sequences at runtime, even though the first is known at compile-time. Worse still, during the process of matching, we cons up lists to hold the variable bindings. If we take advantage of information known at compile-time, we can write a version of `if-match` which performs no unnecessary comparisons, and doesn't cons at all.

If one of the sequences is known at compile-time, and only that one contains variables, then we can go about things differently. In a call to `match`, either argument could contain variables. By restricting variables to the first argument of `if-match`, we make it possible to tell at compile-time which variables will be involved in the match. Then instead of creating lists of variable bindings, we could keep the values of variables in the variables themselves.

The new version of `if-match` appears in Figure 18.7 and 18.8. When we can predict what code would be evaluated at runtime, we can simply generate it at compile-time. Here, instead of expanding into a call to `match`, we generate code which performs just the right comparisons.

If we are going to use the variable `?x` to contain the binding of `?x`, how do we represent a variable for which no binding has yet been established by the match? Here we will indicate that a pattern variable is unbound by binding it to a gensym. So `if-match` begins by generating code which will bind all the variables in the pattern to gensyms. In this case, instead of expanding into a `with-gensyms`, it's safe to make the gensyms once at compile-time and insert them directly into the expansion.

The rest of the expansion is generated by `pat-match`. This macro takes the same arguments as `if-match`; the only difference is that it establishes no new bindings for pattern variables. In some situations this is an advantage, and Chapter 19 will use `pat-match` as an operator in its own right.

In the new matching operator, the distinction between pattern content and pattern structure will be defined by the function `simple?`. If we want to be able to use quoted literals in patterns, the destructuring code (and `vars-in`) have to be told not to go inside lists whose first element is `quote`. With the new matching operator, we will be able to use lists as pattern elements, simply by quoting them.

```
(defmacro if-match (pat seq then &optional else)
  `(let ,(mapcar #'(lambda (v) `(,v ',(gensym)))
                 (vars-in pat #'simple?))
     (pat-match ,pat ,seq ,then ,else)))

(defmacro pat-match (pat seq then else)
  (if (simple? pat)
      (match1 `((,pat ,seq)) then else)
      (with-gensyms (gseq gelse)
        `(labels ((,gelse () ,else))
           ,(gen-match (cons (list gseq seq)
                             (destruc pat gseq #'simple?))
                       then
                       `(,gelse))))))

(defun simple? (x) (or (atom x) (eq (car x) 'quote)))

(defun gen-match (refs then else)
  (if (null refs)
      then
      (let ((then (gen-match (cdr refs) then else)))
        (if (simple? (caar refs))
            (match1 refs then else)
            (gen-match (car refs) then else)))))
```

Figure 18.7: Fast matching operator.

Like dbind, pat-match calls destruc to get a list of the calls that will take apart its argument at runtime. This list is passed on to gen-match, which recursively generates matching code for nested patterns, and thence to match1, which generates match code for each leaf of the pattern tree.

Most of the code which will appear in the expansion of an if-match comes from match1, which is shown in Figure 18.8. This function considers four cases. If the pattern argument is a gensym, then it is one of the invisible variables created by destruc to hold sublists, and all we need to do at runtime is test that it has the right length. If the pattern element is a wildcard (_), no code need be generated. If the pattern element is a variable, match1 generates code to match it against, or set it to, the corresponding part of the sequence given at runtime. Otherwise, the pattern element is taken to be a literal value, and match1 generates code to compare it with the corresponding part of the sequence.

```
(defun match1 (refs then else)
  (dbind ((pat expr) . rest) refs
    (cond ((gensym? pat)
           `(let ((,pat ,expr))
              (if (and (typep ,pat 'sequence)
                       ,(length-test pat rest))
                  ,then
                  ,else)))
          ((eq pat '_) then)
          ((var? pat)
           (let ((ge (gensym)))
             `(let ((,ge ,expr))
                (if (or (gensym? ,pat) (equal ,pat ,ge))
                    (let ((,pat ,ge)) ,then)
                    ,else))))
          (t `(if (equal ,pat ,expr) ,then ,else)))))

(defun gensym? (s)
  (and (symbolp s) (not (symbol-package s))))

(defun length-test (pat rest)
  (let ((fin (caadar (last rest))))
    (if (or (consp fin) (eq fin 'elt))
        `(= (length ,pat) ,(length rest))
        `(> (length ,pat) ,(- (length rest) 2)))))
```

Figure 18.8: Fast matching operator (continued).

Let's look at examples of how some parts of the expansion are generated. Suppose we begin with

```
(if-match (?x 'a) seq
    (print ?x)
    nil)
```

The pattern will be passed to `destruc`, with some gensym (call it g for legibility) to represent the sequence:

```
(destruc '(?x 'a) 'g #'simple?)
```

yielding:

```
((?x (elt g 0)) ((quote a) (elt g 1)))
```

On the front of this list we cons `(g seq)`:

```
((g seq) (?x (elt g 0)) ((quote a) (elt g 1)))
```

and send the whole thing to `gen-match`. Like the naive implementation of `length` (page 22), `gen-match` first recurses all the way to the end of the list, and then builds its return value on the way back up. When it has run out of elements, `gen-match` returns its `then` argument, which will be `?x`. On the way back up the recursion, this return value will be passed as the `then` argument to `match1`. Now we will have a call like:

```
(match1 '(((quote a) (elt g 1))) '(print ?x) '⟨else function⟩)
```

yielding:

```
(if (equal (quote a) (elt g 1))
    (print ?x)
    ⟨else function⟩)
```

This will in turn become the `then` argument to another call to `match1`, the value of which will become the `then` argument of the last call to `match1`. The full expansion of this `if-match` is shown in Figure 18.9.

In this expansion gensyms are used in two completely unrelated ways. The variables used to hold parts of the tree at runtime have gensymed names, in order to avoid capture. And the variable `?x` is initially bound to a gensym, to indicate ○ that it hasn't yet been assigned a value by matching.

In the new `if-match`, the pattern elements are now evaluated instead of being ○ implicitly quoted. This means that Lisp variables can be used in patterns, as well as quoted expressions:

```
> (let ((n 3))
    (if-match (?x n 'n '(a b)) '(1 3 n (a b))
      ?x))
1
```

Two further improvements appear because the new version calls `destruc` (Figure 18.1). The pattern can now contain `&rest` or `&body` keywords (`match` doesn't bother with those). And because `destruc` uses the generic sequence operators `elt` and `subseq`, the new `if-match` will work for any kind of sequence. If `abab` is defined with the new version, it can be used also on vectors and strings:

```
(if-match (?x 'a) seq
    (print ?x))
```

expands into:

```
(let ((?x '#:g1))
  (labels ((#:g3 nil nil))
    (let ((#:g2 seq))
      (if (and (typep #:g2 'sequence)
               (= (length #:g2) 2))
          (let ((#:g5 (elt #:g2 0)))
            (if (or (gensym? x) (equal ?x #:g5))
                (let ((?x #:g5))
                   (if (equal 'a (elt #:g2 1))
                       (print ?x)
                       (#:g3)))
                (#:g3)))
          (#:g3)))))
```

Figure 18.9: Expansion of an `if-match`.

```
> (abab "abab")
#\a
#\b
> (abab #(1 2 1 2))
1
2
```

In fact, patterns can be as complex as patterns to `dbind`:

```
> (if-match (?x (1 . ?y) . ?x) '((a b) #(1 2 3) a b)
      (values ?x ?y))
(A B)
#(2 3)
```

Notice that, in the second return value, the elements of the vector are displayed. To have vectors printed this way, set `*print-array*` to `t`.

In this chapter we are beginning to cross the line into a new kind of programming. We began with simple macros for destructuring. In the final version of `if-match` we have something that looks more like its own language. The remaining chapters describe a whole class of programs which operate on the same philosophy.

19

A Query Compiler

Some of the macros defined in the preceding chapter were large ones. To generate its expansion, `if-match` needed all the code in Figures 18.7 and 18.8, plus `destruc` from Figure 18.1. Macros of this size lead naturally to our last topic, embedded languages. If small macros are extensions to Lisp, large macros define sub-languages within it—possibly with their own syntax or control structure. We saw the beginning of this in `if-match`, which had its own distinct representation for variables.

A language implemented within Lisp is called an *embedded language.* Like "utility," the term is not a precisely defined one; `if-match` probably still counts as a utility, but it is getting close to the borderline.

An embedded language is not a like a language implemented by a traditional compiler or interpreter. It is implemented within some existing language, usually by transformation. There need be no barrier between the base language and the extension: it should be possible to intermingle the two freely. For the implementor, this can mean a huge saving of effort. You can embed just what you need, and for the rest, use the base language.

Transformation, in Lisp, suggests macros. To some extent, you could implement embedded languages with preprocessors. But preprocessors usually operate only on text, while macros take advantage of a unique property of Lisp: between the reader and the compiler, your Lisp program is represented as lists of Lisp objects. Transformations done at this stage can be much smarter.

The best-known example of an embedded language is CLOS, the Common Lisp Object System. If you wanted to make an object-oriented version of a conventional language, you would have to write a new compiler. Not so in Lisp. Tuning the compiler will make CLOS run faster, but in principle the compiler doesn't have to be changed at all. The whole thing can be written in Lisp.

The remaining chapters give examples of embedded languages. This chapter describes how to embed in Lisp a program to answer queries on a database. (You will notice in this program a certain family resemblance to `if-match`.) The first sections describe how to write a system which interprets queries. This program is then reimplemented as a query compiler—in essence, as one big macro—making it both more efficient and better integrated with Lisp.

19.1 The Database

For our present purposes, the format of the database doesn't matter very much. Here, for the sake of convenience, we will store information in lists. For example, we will represent the fact that Joshua Reynolds was an English painter who lived from 1723 to 1792 by:

```
(painter reynolds joshua english)
(dates reynolds 1723 1792)
```

There is no canonical way of reducing information to lists. We could just as well have used one big list:

```
(painter reynolds joshua 1723 1792 english)
```

It is up to the user to decide how to organize database entries. The only restriction is that the entries (*facts*) will be indexed under their first element (the *predicate*). Within those bounds, any consistent form will do, although some forms might make for faster queries than others.

Any database system needs at least two operations: one for modifying the database, and one for examining it. The code shown in Figure 19.1 provides these operations in a basic form. A database is represented as a hash-table filled with lists of facts, hashed according to their predicate.

Although the database functions defined in Figure 19.1 support multiple databases, they all default to operations on `*default-db*`. As with packages in Common Lisp, programs which don't need multiple databases need not even mention them. In this chapter all the examples will just use the `*default-db*`.

We initialize the system by calling `clear-db`, which empties the current database. We can look up facts with a given predicate with `db-query`, and insert new facts into a database entry with `db-push`. As explained in Section 12.1, a macro which expands into an invertible reference will itself be invertible. Since `db-query` is defined this way, we can simply push new facts onto the `db-query` of their predicates. In Common Lisp, hash-table entries are initialized to `nil`

```
(defun make-db (&optional (size 100))
  (make-hash-table :size size))

(defvar *default-db* (make-db))

(defun clear-db (&optional (db *default-db*))
  (clrhash db))

(defmacro db-query (key &optional (db '*default-db*))
  `(gethash ,key ,db))

(defun db-push (key val &optional (db *default-db*))
  (push val (db-query key db)))

(defmacro fact (pred &rest args)
  `(progn (db-push ',pred ',args)
          ',args))
```

Figure 19.1: Basic database functions.

unless specified otherwise, so any key initially has an empty list associated with it. Finally, the macro `fact` adds a new fact to the database.

```
> (fact painter reynolds joshua english)
(REYNOLDS JOSHUA ENGLISH)
> (fact painter canale antonio venetian)
(CANALE ANTONIO VENETIAN)
> (db-query 'painter)
((CANALE ANTONIO VENETIAN)
 (REYNOLDS JOSHUA ENGLISH))
T
```

The `t` returned as the second value by `db-query` appears because `db-query` expands into a `gethash`, which returns as its second value a flag to distinguish between finding no entry and finding an entry whose value is `nil`.

19.2 Pattern-Matching Queries

Calling `db-query` is not a very flexible way of looking at the contents of the database. Usually the user wants to ask questions which depend on more than just the first element of a fact. A *query language* is a language for expressing

⟨query⟩	:	`(`⟨symbol⟩ ⟨argument⟩`*)`
	:	`(not` ⟨query⟩`)`
	:	`(and` ⟨query⟩`*)`
	:	`(or` ⟨query⟩`*)`
⟨argument⟩	:	`?`⟨symbol⟩
	:	⟨symbol⟩
	:	⟨number⟩

Figure 19.2: Syntax of queries.

more complicated questions. In a typical query language, the user can ask for all the values which satisfy some combination of restrictions—for example, the last names of all the painters born in 1697.

Our program will provide a declarative query language. In a declarative query language, the user specifies the constraints which answers must satisfy, and leaves it to the system to figure out how to generate them. This way of expressing queries is close to the form people use in everyday conversation. With our program, we will be able to express the sample query by asking for all the *x* such that there is a fact of the form `(painter` *x* `...)`, and a fact of the form `(dates` *x* `1697 ...)`. We will be able to refer to all the painters born in 1697 by writing:

```
(and (painter ?x ?y ?z)
     (dates ?x 1697 ?w))
```

As well as accepting simple queries consisting of a predicate and some arguments, our program will be able to answer arbitrarily complex queries joined together by logical operators like `and` and `or`. The syntax of the query language is shown in Figure 19.2.

Since facts are indexed under their predicates, variables cannot appear in the predicate position. If you were willing to give up the benefits of indexing, you could get around this restriction by always using the same predicate, and making the first argument the *de facto* predicate.

Like many such systems, this program has a skeptic's notion of truth: some facts are known, and everything else is false. The `not` operator succeeds if the fact in question is not present in the database. To a degree, you could represent explicit falsity by the *Wayne's World* method:

```
(edible motor-oil not)
```

However, the `not` operator wouldn't treat these facts differently from any others.

In programming languages there is a fundamental distinction between interpreted and compiled programs. In this chapter we examine the same question with respect to queries. A query interpreter accepts a query and uses it to generate answers from the database. A query compiler accepts a query and generates a *program* which, when run, yields the same result. The following sections describe a query interpreter and then a query compiler.

19.3 A Query Interpreter

To implement a declarative query language we will use the pattern-matching utilities defined in Section 18.4. The functions shown in Figure 19.3 interpret queries of the form shown in Figure 19.2. The central function in this code is `interpret-query`, which recursively works through the structure of a complex query, generating bindings in the process. The evaluation of complex queries proceeds left-to-right, as in Common Lisp itself.

When the recursion gets down to patterns for facts, `interpret-query` calls `lookup`. This is where the pattern-matching occurs. The function `lookup` takes a pattern consisting of a predicate and a list of arguments, and returns a list of all the bindings which make the pattern match some fact in the database. It gets all the database entries for the predicate, and calls `match` (page 239) to compare each of them against the pattern. Each successful match returns a list of bindings, and `lookup` in turn returns a list of all these lists.

```
> (lookup 'painter '(?x ?y english))
(((?Y . JOSHUA) (?X . REYNOLDS)))
```

These results are then filtered or combined depending on the surrounding logical operators. The final result is returned as a list of sets of bindings. Given the assertions shown in Figure 19.4, here is the example from earlier in this chapter:

```
> (interpret-query '(and (painter ?x ?y ?z)
                         (dates ?x 1697 ?w)))
(((?W . 1768) (?Z . VENETIAN) (?Y . ANTONIO) (?X . CANALE))
 ((?W . 1772) (?Z . ENGLISH) (?Y . WILLIAM) (?X . HOGARTH)))
```

As a general rule, queries can be combined and nested without restriction. In a few cases there are subtle restrictions on the syntax of queries, but these are best dealt with after looking at some examples of how this code is used.

The macro `with-answer` provides a clean way of using the query interpreter within Lisp programs. It takes as its first argument any legal query; the rest of the arguments are treated as a body of code. A `with-answer` expands into

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
    `(dolist (,binds (interpret-query ',query))
       (let ,(mapcar #'(lambda (v)
                         `(,v (binding ',v ,binds)))
                     (vars-in query #'atom))
         ,@body))))

(defun interpret-query (expr &optional binds)
  (case (car expr)
    (and  (interpret-and (reverse (cdr expr)) binds))
    (or   (interpret-or (cdr expr) binds))
    (not  (interpret-not (cadr expr) binds))
    (t    (lookup (car expr) (cdr expr) binds))))

(defun interpret-and (clauses binds)
  (if (null clauses)
      (list binds)
      (mapcan #'(lambda (b)
                  (interpret-query (car clauses) b))
              (interpret-and (cdr clauses) binds))))

(defun interpret-or (clauses binds)
  (mapcan #'(lambda (c)
              (interpret-query c binds))
          clauses))

(defun interpret-not (clause binds)
  (if (interpret-query clause binds)
      nil
      (list binds)))

(defun lookup (pred args &optional binds)
  (mapcan #'(lambda (x)
              (aif2 (match x args binds) (list it)))
          (db-query pred)))
```

Figure 19.3: Query interpreter.

```
(clear-db)
(fact painter hogarth william english)
(fact painter canale antonio venetian)
(fact painter reynolds joshua english)
(fact dates hogarth 1697 1772)
(fact dates canale 1697 1768)
(fact dates reynolds 1723 1792)
```

Figure 19.4: Assertion of sample facts.

code which collects all the sets of bindings generated by the query, then iterates through the body with the variables in the query bound as specified by each set of bindings. Variables which appear in the query of a `with-answer` can (usually) be used within its body. When the query is successful but contains no variables, `with-answer` evaluates the body of code just once.

With the database as defined in Figure 19.4, Figure 19.5 shows some sample queries, accompanied by English translations. Because pattern-matching is done with `match`, it is possible to use the underscore as a wild-card in patterns.

To keep these examples short, the code within the bodies of the queries does nothing more than print some result. In general, the body of a `with-answer` can consist of any Lisp expressions.

19.4 Restrictions on Binding

There are some restrictions on which variables will be bound by a query. For example, why should the query

```
(not (painter ?x ?y ?z))
```

assign any bindings to `?x` and `?y` at all? There are an infinite number of combinations of `?x` and `?y` which are *not* the name of some painter. Thus we add the following restriction: the `not` operator will filter out bindings which are already generated, as in

```
(and (painter ?x ?y ?z) (not (dates ?x 1772 ?d)))
```

but you cannot expect it to generate bindings all by itself. We have to generate sets of bindings by looking for painters before we can screen out the ones not born in 1772. If we had put the clauses in the reverse order:

```
(and (not (dates ?x 1772 ?d)) (painter ?x ?y ?z))     ; wrong
```

The first name and nationality of every painter called Hogarth.

```
> (with-answer (painter hogarth ?x ?y)
    (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

The last name of every painter born in 1697. (Our original example.)

```
> (with-answer (and (painter ?x _ _)
                    (dates ?x 1697 _))
    (princ (list ?x)))
(CANALE)(HOGARTH)
NIL
```

The last name and year of birth of everyone who died in 1772 or 1792.

```
> (with-answer (or (dates ?x ?y 1772)
                   (dates ?x ?y 1792))
    (princ (list ?x ?y)))
(HOGARTH 1697)(REYNOLDS 1723)
NIL
```

The last name of every English painter not born the same year as a Venetian one.

```
> (with-answer (and (painter ?x _ english)
                    (dates ?x ?b _)
                    (not (and (painter ?x2 _ venetian)
                              (dates ?x2 ?b _))))
    (princ ?x))
REYNOLDS
NIL
```

Figure 19.5: The query interpreter in use.

then we would get `nil` as the result if there were *any* painters born in 1772. Even in the first example, we shouldn't expect to be able to use the value of `?d` within the body of a `with-answer` expression.

Also, expressions of the form `(or` $q_1 \ldots q_n$`)` are only guaranteed to generate real bindings for variables which appear in all of the q_i. If a `with-answer` contained the query

```
(or (painter ?x ?y ?z) (dates ?x ?b ?d))
```

you could expect to use the binding of ?x, because no matter which of the subqueries succeeds, it will generate a binding for ?x. But neither ?y nor ?b is guaranteed to get a binding from the query, though one or the other will. Pattern ○ variables not bound by the query will be nil for that iteration.

19.5 A Query Compiler

The code in Figure 19.3 does what we want, but inefficiently. It analyzes the structure of the query at runtime, though it is known at compile-time. And it conses up lists to hold variable bindings, when we could use the variables to hold their own values. Both of these problems can be solved by defining with-answer in a different way.

Figure 19.6 defines a new version of with-answer. The new version continues a trend which began with avg (page 182), and continued with if-match (page 242): it does at compile-time much of the work that the old version did at runtime. The code in Figure 19.6 bears a superficial resemblance to that in Figure 19.3, but none of these functions are called at runtime. Instead of generating bindings, they generate code, which becomes part of the expansion of with-answer. At runtime this code will generate all the bindings which satisfy the query according to the current state of the database.

In effect, this program is one big macro. Figure 19.7 shows the macroexpansion of a with-answer. Most of the work is done by pat-match (page 242), which is itself a macro. Now the only new functions needed at runtime are the basic database functions shown in Figure 19.1.

When with-answer is called from the toplevel, query compilation has little advantage. The code representing the query is generated, evaluated, then thrown away. But when a with-answer expression appears within a Lisp program, the code representing the query becomes part of its macroexpansion. So when the containing program is compiled, the code for all the queries will be compiled inline in the process.

Although the primary advantage of the new approach is speed, it also makes with-answer expressions better integrated with the code in which they appear. This shows in two specific improvements. First, the arguments within the query now get evaluated, so we can say:

```
> (setq my-favorite-year 1723)
1723
> (with-answer (dates ?x my-favorite-year ?d)
    (format t "~A was born in my favorite year.~%" ?x))
REYNOLDS was born in my favorite year.
NIL
```

```
(defmacro with-answer (query &body body)
  `(with-gensyms ,(vars-in query #'simple?)
     ,(compile-query query `(progn ,@body))))

(defun compile-query (q body)
  (case (car q)
    (and  (compile-and (cdr q) body))
    (or   (compile-or  (cdr q) body))
    (not  (compile-not (cadr q) body))
    (lisp `(if ,(cadr q) ,body))
    (t    (compile-simple q body))))

(defun compile-simple (q body)
  (let ((fact (gensym)))
    `(dolist (,fact (db-query ',(car q)))
       (pat-match ,(cdr q) ,fact ,body nil))))

(defun compile-and (clauses body)
  (if (null clauses)
      body
      (compile-query (car clauses)
                     (compile-and (cdr clauses) body))))

(defun compile-or (clauses body)
  (if (null clauses)
      nil
      (let ((gbod (gensym))
            (vars (vars-in body #'simple?)))
        `(labels ((,gbod ,vars ,body))
           ,@(mapcar #'(lambda (cl)
                         (compile-query cl `(,gbod ,@vars)))
                     clauses)))))

(defun compile-not (q body)
  (let ((tag (gensym)))
    `(if (block ,tag
           ,(compile-query q `(return-from ,tag nil))
           t)
         ,body)))
```

Figure 19.6: Query compiler.

```
(with-answer (painter ?x ?y ?z)
  (format t "~A ~A is a painter.~%" ?y ?x))
```

is expanded by the query interpreter into:

```
(dolist (#:g1 (interpret-query '(painter ?x ?y ?z)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1))
        (?z (binding '?z #:g1)))
    (format t "~A ~A is a painter.~%" ?y ?x)))
```

and by the query compiler into:

```
(with-gensyms (?x ?y ?z)
  (dolist (#:g1 (db-query 'painter))
    (pat-match (?x ?y ?z) #:g1
         (progn
           (format t "~A ~A is a painter.~%" ?y ?x))
         nil)))
```

Figure 19.7: Two expansions of the same query.

This could have been done in the query interpreter, but only at the cost of calling `eval` explicitly. And even then, it wouldn't have been possible to refer to lexical variables in the query arguments.

Since arguments within queries are now evaluated, any literal argument (e.g. `english`) that doesn't evaluate to itself should now be quoted. (See Figure 19.8.)

The second advantage of the new approach is that it is now much easier to include normal Lisp expressions within queries. The query compiler adds a `lisp` operator, which may be followed by any Lisp expression. Like the `not` operator, it cannot generate bindings by itself, but it will screen out bindings for which the expression returns `nil`. The `lisp` operator is useful for getting at built-in predicates like `>`:

```
> (with-answer (and (dates ?x ?b ?d)
                    (lisp (> (- ?d ?b) 70)))
    (format t "~A lived over 70 years.~%" ?x))
CANALE lived over 70 years.
HOGARTH lived over 70 years.
NIL
```

A well-implemented embedded language can have a seamless interface with the

The first name and nationality of every painter called Hogarth.

```
> (with-answer (painter 'hogarth ?x ?y)
    (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

The last name of every English painter not born in the same year as a Venetian painter.

```
> (with-answer (and (painter ?x _ 'english)
                    (dates ?x ?b _)
                    (not (and (painter ?x2 _ 'venetian)
                              (dates ?x2 ?b _))))
    (princ ?x))
REYNOLDS
NIL
```

The last name and year of death of every painter who died between 1770 and 1800 exclusive.

```
> (with-answer (and (painter ?x _ _)
                    (dates ?x _ ?d)
                    (lisp (< 1770 ?d 1800)))
    (princ (list ?x ?d)))
(REYNOLDS 1792)(HOGARTH 1772)
NIL
```

Figure 19.8: The query compiler in use.

base language on both sides.

Aside from these two additions—the evaluation of arguments and the new `lisp` operator—the query language supported by the query compiler is identical to that supported by the interpreter. Figure 19.8 shows examples of the results generated by the query compiler with the database as defined in Figure 19.4.

Section 17.2 gave two reasons why it is better to compile an expression than feed it, as a list, to `eval`. The former is faster, and allows the expression to be evaluated in the surrounding lexical context. The advantages of query compilation are exactly analogous. Work that used to be done at runtime is now done at compile-time. And because the queries are compiled as a piece with the surrounding Lisp code, they can take advantage of the lexical context.

20

Continuations

A continuation is a program frozen in action: a single functional object containing the state of a computation. When the object is evaluated, the stored computation is restarted where it left off. In solving certain types of problems it can be a great help to be able to save the state of a program and restart it later. In multiprocessing, for example, a continuation conveniently represents a suspended process. In nondeterministic search programs, a continuation can represent a node in the search tree.

Continuations can be difficult to understand. This chapter approaches the topic in two steps. The first part of the chapter looks at the use of continuations in ○ Scheme, which has built-in support for them. Once the behavior of continuations has been explained, the second part shows how to use macros to build continuations in Common Lisp programs. Chapters 21–24 will all make use of the macros defined here.

20.1 Scheme Continuations

One of the principal ways in which Scheme differs from Common Lisp is its explicit support for continuations. This section shows how continuations work in Scheme. (Figure 20.1 lists some other differences between Scheme and Common Lisp.)

A continuation is a function representing the future of a computation. Whenever an expression is evaluated, something is waiting for the value it will return. For example, in

1. Scheme makes no distinction between what Common Lisp calls the `symbol-value` and `symbol-function` of a symbol. In Scheme, a variable has a single value, which can be either a function or some other sort of object. Thus there is no need for sharp-quote or `funcall` in Scheme. The Common Lisp:

```
(let ((f #'(lambda (x) (1+ x))))
  (funcall f 2))
```

would be in Scheme:

```
(let ((f (lambda (x) (1+ x))))
  (f 2))
```

2. Since Scheme has only one name-space, it doesn't need separate operators (e.g. `defun` and `setq`) for assignments in each. Instead it has `define`, which is roughly equivalent to `defvar`, and `set!`, which takes the place of `setq`. Global variables must be created with `define` before they can be set with `set!`.

3. In Scheme, named functions are usually defined with `define`, which takes the place of `defun` as well as `defvar`. The Common Lisp:

```
(defun foo (x) (1+ x))
```

has two possible Scheme translations:

```
(define foo (lambda (x) (1+ x)))
(define (foo x) (1+ x))
```

4. In Common Lisp, the arguments to a function are evaluated left-to-right. In Scheme, the order of evaluation is deliberately unspecified. (And implementors delight in surprising those who forget this.)

5. Instead of `t` and `nil`, Scheme has `#t` and `#f`. The empty list, `()`, is true in some implementations and false in others.

6. The default clause in `cond` and `case` expressions has the key `else` in Scheme, instead of `t` as in Common Lisp.

7. Several built-in operators have different names: `consp` is `pair?`, `null` is `null?`, `mapcar` is (almost) `map`, and so on. Ordinarily these should be obvious from the context.

Figure 20.1: Some differences between Scheme and Common Lisp.

```
(/ (- x 1) 2)
```

when `(- x 1)` is evaluated, the outer `/` expression is waiting for the value, and something else is waiting for *its* value, and so on and so on, all the way back to the toplevel—where `print` is waiting.

We can think of the continuation at any given time as a function of one argument. If the previous expression were typed into the toplevel, then when the subexpression `(- x 1)` was evaluated, the continuation would be:

```
(lambda (val) (/ val 2))
```

That is, the remainder of the computation could be duplicated by calling this function on the return value. If instead the expression occurred in the following context

```
(define (f1 w)
  (let ((y (f2 w)))
    (if (integer? y) (list 'a y) 'b)))

(define (f2 x)
  (/ (- x 1) 2))
```

and `f1` were called from the toplevel, then when `(- x 1)` was evaluated, the continuation would be equivalent to

```
(lambda (val)
  (let ((y (/ val 2)))
    (if (integer? y) (list 'a y) 'b)))
```

In Scheme, continuations are first-class objects, just like functions. You can ask Scheme for the current continuation, and it will make you a function of one argument representing the future of the computation. You can save this object for as long as you like, and when you call it, it will restart the computation that was taking place when it was created.

Continuations can be understood as a generalization of closures. A closure is a function plus pointers to the lexical variables visible at the time it was created. A continuation is a function plus a pointer to the whole stack pending at the time it was created. When a continuation is evaluated, it returns a value using its own copy of the stack, ignoring the current one. If a continuation is created at T_1 and evaluated at T_2, it will be evaluated with the stack that was pending at T_1.

Scheme programs have access to the current continuation via the built-in operator `call-with-current-continuation` (`call/cc` for short). When a program calls `call/cc` on a function of one argument:

```
(call-with-current-continuation
  (lambda (cc)
    ...))
```

the function will be passed another function representing the current continuation. By storing the value of `cc` somewhere, we save the state of the computation at the point of the `call/cc`.

In this example, we `append` together a list whose last element is the value returned by a `call/cc` expression:

```
> (define frozen)
FROZEN
> (append '(the call/cc returned)
          (list (call-with-current-continuation
                  (lambda (cc)
                    (set! frozen cc)
                    'a))))
(THE CALL/CC RETURNED A)
```

The `call/cc` returns `a`, but first saves the continuation in the global variable `frozen`.

Calling `frozen` will restart the old computation at the point of the `call/cc`. Whatever value we pass to `frozen` will be returned as the value of the `call/cc`:

```
> (frozen 'again)
(THE CALL/CC RETURNED AGAIN)
```

Continuations aren't used up by being evaluated. They can be called repeatedly, just like any other functional object:

```
> (frozen 'thrice)
(THE CALL/CC RETURNED THRICE)
```

When we call a continuation within some other computation, we see more clearly what it means to return back up the old stack:

```
> (+ 1 (frozen 'safely))
(THE CALL/CC RETURNED SAFELY)
```

Here, the pending `+` is ignored when `frozen` is called. The latter returns up the stack that was pending at the time it was first created: through `list`, then `append`, to the toplevel. If `frozen` returned a value like a normal function call, the expression above would have yielded an error when `+` tried to add `1` to a list.

Continuations do not get unique copies of the stack. They may share variables with other continuations, or with the computation currently in progress. In this example, two continuations share the same stack:

```
> (define froz1)
FROZ1
> (define froz2)
FROZ2
> (let ((x 0))
    (call-with-current-continuation
      (lambda (cc)
        (set! froz1 cc)
        (set! froz2 cc)))
    (set! x (1+ x))
    x)
1
```

so calls to either will return successive integers:

```
> (froz2 ())
2
> (froz1 ())
3
```

Since the value of the `call/cc` expression will be discarded, it doesn't matter what argument we give to `froz1` and `froz2`.

Now that we can store the state of a computation, what do we do with it? Chapters 21–24 are devoted to applications which use continuations. Here we will consider a simple example which conveys well the flavor of programming with saved states: we have a set of trees, and we want to generate lists containing one element from each tree, until we get a combination satisfying some condition.

Trees can be represented as nested lists. Page 70 described a way to represent one kind of tree as a list. Here we use another, which allows interior nodes to have (atomic) values, and any number of children. In this representation, an interior node becomes a list; its car contains the value stored at the node, and its cdr contains the representations of the node's children. For example, the two trees shown in Figure 20.2 can be represented:

```
(define t1 '(a (b (d h)) (c e (f i) g)))
(define t2 '(1 (2 (3 6 7) 4 5)))
```

Figure 20.3 contains functions which do depth-first traversals on such trees. In a real program we would want to do something with the nodes as we encountered them. Here we just print them. The function `dft`, given for comparison, does an ordinary depth-first traversal:

```
> (dft t1)
ABDHCEFIG()
```

Figure 20.2: Two Trees.

The function `dft-node` follows the same path through the tree, but deals out nodes one at a time. When `dft-node` reaches a node, it follows the car of the node, and pushes onto `*saved*` a continuation to explore the cdr.

```
> (dft-node t1)
A
```

Calling `restart` continues the traversal, by popping the most recently saved continuation and calling it.

```
> (restart)
B
```

Eventually there will be no saved states left, a fact which `restart` signals by returning `done`:

```
⋮
> (restart)
G
> (restart)
DONE
```

Finally, the function `dft2` neatly packages up what we just did by hand:

```
> (dft2 t1)
ABDHCEFIG()
```

```
(define (dft tree)
  (cond ((null? tree) ())
        ((not (pair? tree)) (write tree))
        (else (dft (car tree))
              (dft (cdr tree)))))

(define *saved* ())

(define (dft-node tree)
  (cond ((null? tree) (restart))
        ((not (pair? tree)) tree)
        (else (call-with-current-continuation
                (lambda (cc)
                  (set! *saved*
                        (cons (lambda ()
                                (cc (dft-node (cdr tree))))
                              *saved*))
                  (dft-node (car tree)))))))

(define (restart)
  (if (null? *saved*)
      'done
      (let ((cont (car *saved*)))
        (set! *saved* (cdr *saved*))
        (cont))))

(define (dft2 tree)
  (set! *saved* ())
  (let ((node (dft-node tree)))
    (cond ((eq? node 'done) ())
          (else (write node)
                (restart)))))
```

Figure 20.3: Tree traversal using continuations.

Notice that there is no explicit recursion or iteration in the definition of `dft2`: successive nodes are printed because the continuations invoked by `restart` always return back through the same `cond` clause in `dft-node`.

This kind of program works like a mine. It digs the initial shaft by calling `dft-node`. So long as the value returned is not `done`, the code following the call to `dft-node` will call `restart`, which sends control back down the stack again. This process continues until the return value signals that the mine is empty. Instead of printing this value, `dft2` returns `#f`. Search with continuations represents a novel way of thinking about programs: put the right code in the stack, and get the result by repeatedly returning up through it.

If we only want to traverse one tree at a time, as in `dft2`, then there is no reason to bother using this technique. The advantage of `dft-node` is that we can have several instances of it going at once. Suppose we have *two* trees, and we want to generate, in depth-first order, the cross-product of their elements.

```
> (set! *saved* ())
()
> (let ((node1 (dft-node t1)))
    (if (eq? node1 'done)
        'done
        (list node1 (dft-node t2))))
(A 1)
> (restart)
(A 2)
⋮
> (restart)
(B 1)
⋮
```

Using normal techniques, we would have had to take explicit steps to save our place in the two trees. With continuations, the state of the two ongoing traversals is maintained automatically. In a simple case like this one, saving our place in the tree would not be so difficult. The trees are permanent data structures, so at least we have some way of getting hold of "our place" in the tree. The great thing about continuations is that they can just as easily save our place in the middle of *any* computation, even if there are no permanent data structures associated with it. The computation need not even have a finite number of states, so long as we only want to restart a finite number of them.

As Chapter 24 will show, both of these considerations turn out to be important in the implementation of Prolog. In Prolog programs, the "search trees" are not real data structures, but are implicit in the way the program generates results. And the trees are often infinite, in which case we cannot hope to search the whole of one before searching the next; we have no choice but to save our place, one way or another.

20.2 Continuation-Passing Macros

Common Lisp doesn't provide `call/cc`, but with a little extra effort we can do the same things as we can in Scheme. This section shows how to use macros to build continuations in Common Lisp programs. Scheme continuations gave us two things:

1. The bindings of all variables at the time the continuation was made.

2. The state of the computation—what was going to happen from then on.

In a lexically scoped Lisp, closures give us the first of these. It turns out that we can also use closures to maintain the second, by storing the state of the computation in variable bindings as well.

The macros shown in Figure 20.4 make it possible to do function calls while preserving continuations. These macros replace the built-in Common Lisp forms for defining functions, calling them, and returning values.

Functions which want to use continuations (or call functions which do) should be defined with `=defun` instead of `defun`. The syntax of `=defun` is the same as that of `defun`, but its effect is subtly different. Instead of defining just a function, `=defun` defines a function and a macro which expands into a call to it. (The macro must be defined first, in case the function calls itself.) The function will have the body that was passed to `=defun`, but will have an additional parameter, `*cont*`, consed onto its parameter list. In the expansion of the macro, this function will receive `*cont*` along with its other arguments. So

```
(=defun add1 (x) (=values (1+ x)))
```

macroexpands into

```
(progn (defmacro add1 (x)
         `(=add1 *cont* ,x))
       (defun =add1 (*cont* x)
         (=values (1+ x))))
```

When we call `add1`, we are actually calling not a function but a macro. The macro expands into a function call,[1] but with one extra parameter: `*cont*`. So the current value of `*cont*` is always passed implicitly in a call to an operator defined with `=defun`.

What is `*cont*` for? It will be bound to the current continuation. The definition of `=values` shows how this continuation will be used. Any function defined using `=defun` must return with `=values`, or call some other function

[1] Functions created by `=defun` are deliberately given interned names, to make it possible to `trace` them. If tracing were never necessary, it would be safer to gensym the names.

```
(setq *cont* #'identity)

(defmacro =lambda (parms &body body)
  `#'(lambda (*cont* ,@parms) ,@body))

(defmacro =defun (name parms &body body)
  (let ((f (intern (concatenate 'string
                                "=" (symbol-name name)))))
    `(progn
       (defmacro ,name ,parms
         `(,',f *cont* ,,@parms))
       (defun ,f (*cont* ,@parms) ,@body))))

(defmacro =bind (parms expr &body body)
  `(let ((*cont* #'(lambda ,parms ,@body))) ,expr))

(defmacro =values (&rest retvals)
  `(funcall *cont* ,@retvals))

(defmacro =funcall (fn &rest args)
  `(funcall ,fn *cont* ,@args))

(defmacro =apply (fn &rest args)
  `(apply ,fn *cont* ,@args))
```

Figure 20.4: Continuation-passing macros.

which does so. The syntax of `=values` is the same as that of the Common Lisp form `values`. It can return multiple values if there is an `=bind` with the same number of arguments waiting for them, but can't return multiple values to the toplevel. ○

The parameter `*cont*` tells a function defined with `=defun` what to do with its return value. When `=values` is macroexpanded it will capture `*cont*`, and use it to simulate returning from the function. The expression

```
> (=values (1+ n))
```

expands into

```
(funcall *cont* (1+ n))
```

At the toplevel, the value of `*cont*` is `identity`, which just returns whatever is passed to it. When we call `(add1 2)` from the toplevel, the call gets macroexpanded into the equivalent of

```
(funcall #'(lambda (*cont* n) (=values (1+ n))) *cont* 2)
```

The reference to `*cont*` will in this case get the global binding. The `=values` expression will thus macroexpand into the equivalent of:

```
(funcall #'identity (1+ n))
```

which just adds `1` to `n` and returns the result.

In functions like `add1`, we go through all this trouble just to simulate what Lisp function call and return do anyway:

```
> (=defun bar (x)
    (=values (list 'a (add1 x))))
BAR
> (bar 5)
(A 6)
```

The point is, we have now brought function call and return under our own control, and can do other things if we wish.

It is by manipulating `*cont*` that we will get the effect of continuations. Although `*cont*` has a global value, this will rarely be the one used: `*cont*` will nearly always be a parameter, captured by `=values` and the macros defined by `=defun`. Within the body of `add1`, for example, `*cont*` is a parameter and not the global variable. This distinction is important because these macros wouldn't work if `*cont*` were not a local variable. That's why `*cont*` is given its initial value in a `setq` instead of a `defvar`: the latter would also proclaim it to be `special`.

The third macro in Figure 20.4, `=bind`, is intended to be used in the same way as `multiple-value-bind`. It takes a list of parameters, an expression, and a body of code: the parameters are bound to the values returned by the expression, and the code body is evaluated with those bindings. This macro should be used whenever additional expressions have to be evaluated after calling a function defined with `=defun`.

```
> (=defun message ()
    (=values 'hello 'there))
MESSAGE
```

```
> (=defun baz ()
    (=bind (m n) (message)
      (=values (list m n))))
BAZ
> (baz)
(HELLO THERE)
```

Notice that the expansion of an `=bind` creates a new variable called `*cont*`. The body of `baz` macroexpands into:

```
(let ((*cont* #'(lambda (m n)
                  (=values (list m n)))))
  (message))
```

which in turn becomes:

```
(let ((*cont* #'(lambda (m n)
                  (funcall *cont* (list m n)))))
  (=message *cont*))
```

The new value of `*cont*` is the body of the `=bind` expression, so when `message` "returns" by funcalling `*cont*`, the result will be to evaluate the body of code. However (and this is the key point), within the body of the `=bind`:

```
#'(lambda (m n)
    (funcall *cont* (list m n)))
```

the `*cont*` that was passed as an argument to `=baz` is still visible, so when the body of code in turn evaluates an `=values`, *it* will be able to return to the original calling function. The closures are knitted together: each binding of `*cont*` is a closure containing the previous binding of `*cont*`, forming a chain which leads all the way back up to the global value.

We can see the same phenomenon on a smaller scale here:

```
> (let ((f #'identity))
    (let ((g #'(lambda (x) (funcall f (list 'a x)))))
      #'(lambda (x) (funcall g (list 'b x)))))
#<Interpreted-Function BF6326>
> (funcall * 2)
(A (B 2))
```

This example creates a function which is a closure containing a reference to `g`, which is itself a closure containing a reference to `f`. Similar chains of closures were built by the network compiler on page 80.

1. The parameter list of a function defined with `=defun` must consist solely of parameter names.

2. Functions which make use of continuations, or call other functions which do, must be defined with `=lambda` or `=defun`.

3. Such functions must terminate either by returning values with `=values`, or by calling another function which obeys this restriction.

4. If an `=bind`, `=values`, `=apply`, or `=funcall` expression occurs in a segment of code, it must be a tail call. Any code to be evaluated after an `=bind` should be put in its body. So if we want to have several `=binds` one after another, they must be nested:

```
(=defun foo (x)
  (=bind (y) (bar x)
    (format t "Ho ")
    (=bind (z) (baz x)
      (format t "Hum.")
      (=values x y z))))
```

Figure 20.5: Restrictions on continuation-passing macros.

The remaining macros, `=apply` and `=funcall`, are for use with functions defined by `=lambda`. Note that "functions" defined with `=defun`, because they are actually macros, cannot be given as arguments to `apply` or `funcall`. The way around this problem is analogous to the trick mentioned on page 110. It is to package up the call inside another `=lambda`:

```
> (=defun add1 (x)
    (=values (1+ x)))
ADD1
> (let ((fn (=lambda (n) (add1 n))))
    (=bind (y) (=funcall fn 9)
      (format nil "9 + 1 = ~A" y)))
"9 + 1 = 10"
```

Figure 20.5 summarizes all the restrictions imposed by the continuation-passing macros. Functions which neither save continuations, nor call other functions which do, need not use these special macros. Built-in functions like `list`, for example, are exempt.

Figure 20.6 contains the code from Figure 20.3, translated from Scheme into Common Lisp, and using the continuation-passing macros instead of Scheme

```
(defun dft (tree)
  (cond ((null tree) nil)
        ((atom tree) (princ tree))
        (t (dft (car tree))
           (dft (cdr tree)))))

(setq *saved* nil)

(=defun dft-node (tree)
  (cond ((null tree) (restart))
        ((atom tree) (=values tree))
        (t (push #'(lambda () (dft-node (cdr tree)))
                 *saved*)
           (dft-node (car tree)))))

(=defun restart ()
  (if *saved*
      (funcall (pop *saved*))
      (=values 'done)))

(=defun dft2 (tree)
  (setq *saved* nil)
  (=bind (node) (dft-node tree)
    (cond ((eq node 'done) (=values nil))
          (t (princ node)
             (restart)))))
```

Figure 20.6: Tree traversal using continuation-passing macros.

continuations. With the same example tree, `dft2` works just as before:

```
> (setq t1 '(a (b (d h)) (c e (f i) g))
        t2 '(1 (2 (3 6 7) 4 5)))
(1 (2 (3 6 7) 4 5))
> (dft2 t1)
ABDHCEFIG
NIL
```

Saving states of multiple traversals also works as in Scheme, though the example becomes a bit longer:

```
> (=bind (node1) (dft-node t1)
    (if (eq node1 'done)
        'done
        (=bind (node2) (dft-node t2)
          (list node1 node2))))
(A 1)
> (restart)
(A 2)
⋮
> (restart)
(B 1)
⋮
```

By knitting together a chain of lexical closures, Common Lisp programs can build their own continuations. Fortunately, the closures are knitted together by the macros in the sweatshop of Figure 20.4, and the user can have the finished garment without giving a thought to its origins.

Chapters 21–24 all rely on continuations in some way. These chapters will show that continuations are an abstraction of unusual power. They may not be overly fast, especially when implemented on top of the language as macros, but the abstractions we can build upon them make certain programs much faster to write, and there is a place for that kind of speed too.

20.3 Code-Walkers and CPS Conversion

The macros described in the previous section represent a compromise. They give us the power of continuations, but only if we write our programs in a certain way. Rule 4 in Figure 20.5 means that we always have to write

```
(=bind (x) (fn y)
  (list 'a x))
```

rather than

```
(list 'a                                              ; wrong
      (=bind (x) (fn y) x))
```

A true `call/cc` imposes no such restrictions on the programmer. A `call/cc` can grab the continuation at any point in a program of any shape. We could implement an operator with the full power of `call/cc`, but it would be a lot more work. This section outlines how it could be done.

A Lisp program can be transformed into a form called "continuation-passing style." Programs which have undergone complete CPS conversion are impossible to read, but one can grasp the spirit of this process by looking at code which has been partially transformed. The following function to reverse lists:

```
(defun rev (x)
  (if (null x)
      nil
      (append (rev (cdr x)) (list (car x)))))
```

yields an equivalent continuation-passing version:

```
(defun rev2 (x)
  (revc x #'identity))

(defun revc (x k)
  (if (null x)
      (funcall k nil)
      (revc (cdr x)
            #'(lambda (w)
                (funcall k (append w (list (car x))))))))
```

In the continuation-passing style, functions get an additional parameter (here `k`) whose value will be the continuation. The continuation is a closure representing what should be done with the current value of the function. On the first recursion, the continuation is `identity`; what should be done is that the function should just return its current value. On the second recursion, the continuation will be equivalent to:

```
#'(lambda (w)
    (identity (append w (list (car x)))))
```

which says that what should be done is to append the car of the list to the current value, and return it.

Once you can do CPS conversion, it is easy to write `call/cc`. In a program which has undergone CPS conversion, the entire current continuation is always present, and `call/cc` can be implemented as a simple macro which calls some function with it as an argument.

To do CPS conversion we need a *code-walker,* a program that traverses the trees representing the source code of a program. Writing a code-walker for Common Lisp is a serious undertaking. To be useful, a code-walker has to do more than simply traverse expressions. It also has to know a fair amount about what the expressions mean. A code-walker can't just think in terms of symbols, for example. A symbol could represent, among other things, itself, a function, a variable, a block name, or a tag for `go`. The code-walker has to use the context to distinguish one kind of symbol from another, and act accordingly.

Since writing a code-walker would be beyond the scope of this book, the macros described in this chapter are the most practical alternative. The macros in this chapter split the work of building continuations with the user. If the user writes programs in something sufficiently close to CPS, the macros can do the rest. That's what rule 4 really amounts to: if everything following an `=bind` expression is within its body, then between the value of `*cont*` and the code in the body of the `=bind`, the program has enough information to construct the current continuation.

The `=bind` macro is deliberately written to make this style of programming feel natural. In practice the restrictions imposed by the continuation-passing macros are bearable.

21

Multiple Processes

The previous chapter showed how continuations allow a running program to get hold of its own state, and store it away to be restarted later. This chapter deals with a model of computation in which a computer runs not one single *program,* but a collection of independent *processes.* The concept of a process corresponds closely with our concept of the state of a program. By writing an additional layer of macros on top of those in the previous chapter, we can embed multiprocessing in Common Lisp programs.

21.1 The Process Abstraction

Multiple processes are a convenient way of expressing programs which must do several things at once. A traditional processor executes one instruction at a time. To say that multiple processes do more than one thing at once is not to say that they somehow overcome this hardware limitation: what it means is that they allow us to think at a new level of abstraction, in which we don't have to specify exactly what the computer is doing at any given time. Just as virtual memory allows us to act as though the computer had more memory than it actually does, the notion of a process allows us to act as if the computer could run more than one program at a time.

The study of processes is traditionally in the domain of operating systems. But the usefulness of processes as an abstraction is not limited to operating systems. They are equally useful in other real-time applications, and in simulations.

Much of the work done on multiple processes has been devoted to avoiding certain types of problems. Deadlock is one classic problem with multiple processes: two processes both stand waiting for the other to do something, like two people who each refuse to cross a threshold before the other. Another problem is the query which catches the system in an inconsistent state—say, a balance inquiry which arrives while the system is transferring funds from one account to another. This chapter deals only with the process abstraction itself; the code presented here could be used to test algorithms for preventing deadlock or inconsistent states, but it does not itself provide any protection against these problems.

The implementation in this chapter follows a rule implicit in all the programs in this book: disturb Lisp as little as possible. In spirit, a program ought to be as much as possible like a modification of the language, rather than a separate application written in it. Making programs harmonize with Lisp makes them more robust, like a machine whose parts fit together well. It also saves effort; sometimes you can make Lisp do a surprising amount of your work for you.

The aim of this chapter is to make a language which supports multiple processes. Our strategy will be to turn Lisp into such a language, by adding a few new operators. The basic elements of our language will be as follows:

> *Functions* will be defined with the `=defun` or `=lambda` macros from the previous chapter.
>
> *Processes* will be instantiated from function calls. There is no limit on the number of active processes, or the number of processes instantiated from any one function. Each process will have a priority, initially given as an argument when it is created.
>
> *Wait expressions* may occur within functions. A wait expression will take a variable, a test expression, and a body of code. If a process encounters a wait, the process will be suspended at that point until the test expression returns true. Once the process restarts, the body of code will be evaluated, with the variable bound to the value of the test expression. Test expressions should not ordinarily have side-effects, because there are no guarantees about when, or how often, they will be evaluated.
>
> *Scheduling* will be done by priority. Of all the processes able to restart, the system will run the one with the highest priority.
>
> *The default process* will run if no other process can. It is a read-eval-print loop.
>
> *Creation and deletion* of most objects will be possible on the fly. From running processes it will be possible to define new functions, and to instantiate and kill processes.

```
(defstruct proc  pri state wait)

(proclaim '(special *procs* *proc*))

(defvar *halt* (gensym))

(defvar *default-proc*
        (make-proc :state #'(lambda (x)
                              (format t "~%>> ")
                              (princ (eval (read)))
                              (pick-process))))

(defmacro fork (expr pri)
  `(prog1 ',expr
          (push (make-proc
                  :state #'(lambda (,(gensym))
                             ,expr
                             (pick-process))
                  :pri   ,pri)
                *procs*)))

(defmacro program (name args &body body)
  `(=defun ,name ,args
     (setq *procs* nil)
     ,@body
     (catch *halt* (loop (pick-process)))))
```

Figure 21.1: Process structure and instantiation.

Continuations make it possible to store the state of a Lisp program. Being able to store several states at once is not very far from having multiple processes. Starting with the macros defined in the previous chapter, we need less than 60 lines of code to implement multiple processes.

21.2 Implementation

Figures 21.1 and 21.2 contain all the code needed to support multiple processes. Figure 21.1 contains code for the basic data structures, the default process, initialization, and instantiation of processes. Processes, or `procs`, have the following structure:

`pri` is the priority of the process, which should be a positive number.

`state` is a continuation representing the state of a suspended process. A process is restarted by funcalling its `state`.

`wait` is usually a function which must return true in order for the process to be restarted, but initially the `wait` of a newly created process is `nil`. A process with a null `wait` can always be restarted.

The program uses three global variables: `*procs*`, the list of currently suspended processes; `*proc*`, the process now running; and `*default-proc*`, the default process.

The default process runs only when no other process can. It simulates the Lisp toplevel. Within this loop, the user can halt the program, or type expressions which enable suspended processes to restart. Notice that the default process calls `eval` explicitly. This is one of the few situations in which it is legitimate to do so. Generally it is not a good idea to call `eval` at runtime, for two reasons:

1. It's inefficient: `eval` is handed a raw list, and either has to compile it on the spot, or evaluate it in an interpreter. Either way is slower than compiling the code beforehand, and just calling it.

2. It's less powerful, because the expression is evaluated with no lexical context. Among other things, this means that you can't refer to ordinary variables visible outside the expression being evaluated.

Usually, calling `eval` explicitly is like buying something in an airport gift-shop. Having waited till the last moment, you have to pay high prices for a limited selection of second-rate goods.

Cases like this are rare instances when neither of the two preceding arguments applies. We couldn't possibly have compiled the expressions beforehand. We are just now reading them; there is no beforehand. Likewise, the expression can't refer to surrounding lexical variables, because expressions typed at the toplevel are in the null lexical environment. In fact, the definition of this function simply reflects its English description: it reads and evaluates what the user types.

The macro `fork` instantiates a process from a function call. Functions are defined as usual with `=defun`:

```
(=defun foo (x)
  (format t "Foo was called with ~A.~%" x)
  (=values (1+ x)))
```

Now when we call `fork` with a function call and a priority number:

```
(fork (foo 2) 25)
```

a new process is pushed onto `*procs*`. The new process has a priority of `25`, a `proc-wait` of `nil`, since it hasn't been started yet, and a `proc-state` consisting of a call to `foo` with the argument `2`.

The macro `program` allows us to create a group of processes and run them together. The definition:

```
(program two-foos (a b)
  (fork (foo a) 99)
  (fork (foo b) 99))
```

macroexpands into the two `fork` expressions, sandwiched between code which clears out the suspended processes, and other code which repeatedly chooses a process to run. Outside this loop, the macro establishes a tag to which control can be thrown to end the program. As a gensym, this tag will not conflict with tags established by user code. A group of processes defined as a `program` returns no particular value, and is only meant to be called from the toplevel.

After the processes are instantiated, the process scheduling code takes over. This code is shown in Figure 21.2. The function `pick-process` selects and runs the highest priority process which is able to restart. Selecting this process is the job of `most-urgent-process`. A suspended process is eligible to run if it has no `wait` function, or its `wait` function returns true. Among eligible processes, the one with the highest priority is chosen. The winning process and the value returned by its `wait` function (if there is one) are returned to `pick-process`. There will always be some winning process, because the default process always wants to run.

The remainder of the code in Figure 21.2 defines the operators used to switch control between processes. The standard wait expression is `wait`, as used in the function `pedestrian` in Figure 21.3. In this example, the process waits until there is something in the list `*open-doors*`, then prints a message:

```
> (ped)
>> (push 'door2 *open-doors*)
Entering DOOR2
>> (halt)
NIL
```

A `wait` is similar in spirit to an `=bind` (page 267), and carries the same restriction that it must be the last thing to be evaluated. Anything we want to happen after the `wait` must be put in its body. Thus, if we want to have a process wait several times, the wait expressions must be nested. By asserting facts aimed at one another, processes can cooperate in reaching some goal, as in Figure 21.4.

```
(defun pick-process ()
  (multiple-value-bind (p val) (most-urgent-process)
    (setq *proc*  p
          *procs* (delete p *procs*))
    (funcall (proc-state p) val)))

(defun most-urgent-process ()
  (let ((proc1 *default-proc*) (max -1) (val1 t))
    (dolist (p *procs*)
      (let ((pri (proc-pri p)))
        (if (> pri max)
            (let ((val (or (not (proc-wait p))
                           (funcall (proc-wait p)))))
              (when val
                (setq proc1 p
                      max   pri
                      val1  val))))))
    (values proc1 val1)))

(defun arbitrator (test cont)
  (setf (proc-state *proc*) cont
        (proc-wait *proc*)  test)
  (push *proc* *procs*)
  (pick-process))

(defmacro wait (parm test &body body)
  `(arbitrator #'(lambda () ,test)
               #'(lambda (,parm) ,@body)))

(defmacro yield (&body body)
  `(arbitrator nil #'(lambda (,(gensym)) ,@body)))

(defun setpri (n) (setf (proc-pri *proc*) n))

(defun halt (&optional val) (throw *halt* val))

(defun kill (&optional obj &rest args)
  (if obj
      (setq *procs* (apply #'delete obj *procs* args))
      (pick-process)))
```

Figure 21.2: Process scheduling.

```
(defvar *open-doors* nil)

(=defun pedestrian ()
  (wait d (car *open-doors*)
    (format t "Entering ~A~%" d)))

(program ped ()
  (fork (pedestrian) 1))
```

Figure 21.3: One process with one wait.

Processes instantiated from `visitor` and `host`, if given the same `door`, will exchange control via messages on a blackboard:

```
> (ballet)
Approach DOOR2. Open DOOR2. Enter DOOR2. Close DOOR2.
Approach DOOR1. Open DOOR1. Enter DOOR1. Close DOOR1.
>>
```

There is another, simpler type of wait expression: `yield`, whose only purpose is to give other higher-priority processes a chance to run. A process might want to yield after executing a `setpri` expression, which resets the priority of the current process. As with a `wait`, any code to be executed after a `yield` must be put within its body.

The program in Figure 21.5 illustrates how the two operators work together. Initially, the barbarians have two aims: to capture Rome and to plunder it. Capturing the city has (slightly) higher priority, and so will run first. However, after the city has been reduced, the priority of the `capture` process decreases to `1`. Then there is a vote, and `plunder`, as the highest-priority process, starts running.

```
> (barbarians)
Liberating ROME.
Nationalizing ROME.
Refinancing ROME.
Rebuilding ROME.
>>
```

Only after the barbarians have looted Rome's palaces and ransomed the patricians, does the `capture` process resume, and the barbarians turn to fortifying their own position.

Underlying wait expressions is the more general `arbitrator`. This function stores the current process, and then calls `pick-process` to start some process

```
(defvar *bboard* nil)

(defun claim   (&rest f) (push f *bboard*))

(defun unclaim (&rest f) (pull f *bboard* :test #'equal))

(defun check   (&rest f) (find f *bboard* :test #'equal))

(=defun visitor (door)
  (format t "Approach ~A. " door)
  (claim 'knock door)
  (wait d (check 'open door)
    (format t "Enter ~A. " door)
    (unclaim 'knock door)
    (claim 'inside door)))

(=defun host (door)
  (wait k (check 'knock door)
    (format t "Open ~A. " door)
    (claim 'open door)
    (wait g (check 'inside door)
      (format t "Close ~A.~%" door)
      (unclaim 'open door))))

(program ballet ()
  (fork (visitor 'door1) 1)
  (fork (host 'door1) 1)
  (fork (visitor 'door2) 1)
  (fork (host 'door2) 1))
```

Figure 21.4: Synchronization with a blackboard.

(perhaps the same one) running again. It will be given two arguments: a test function and a continuation. The former will be stored as the `proc-wait` of the process being suspended, and called later to determine if it can be restarted. The latter will become the `proc-state`, and calling it will restart the suspended process.

The macros `wait` and `yield` build this continuation function simply by wrapping their bodies in lambda-expressions. For example,

```
(wait d (car *bboard*) (=values d))
```

```
(=defun capture (city)
  (take city)
  (setpri 1)
  (yield
    (fortify city)))

(=defun plunder (city)
  (loot city)
  (ransom city))

(defun take (c)    (format t "Liberating ~A.~%" c))
(defun fortify (c) (format t "Rebuilding ~A.~%" c))
(defun loot (c)    (format t "Nationalizing ~A.~%" c))
(defun ransom (c)  (format t "Refinancing ~A.~%" c))

(program barbarians ()
  (fork (capture 'rome) 100)
  (fork (plunder 'rome) 98))
```

Figure 21.5: Effect of changing priorities.

expands into:

```
(arbitrator #'(lambda () (car *bboard*))
            #'(lambda (d) (=values d)))
```

If the code obeys the restrictions listed in Figure 20.5, making a closure of the `wait`'s body will preserve the whole current continuation. With its `=values` expanded the second argument becomes:

```
#'(lambda (d) (funcall *cont* d))
```

Since the closure contains a reference to `*cont*`, the suspended process with this wait function will have a handle on where it was headed at the time it was suspended.

The `halt` operator stops the whole program, by throwing control back to the tag established by the expansion of `program`. It takes an optional argument, which will be returned as the value of the program. Because the default process is always willing to run, the only way programs end is by explicit `halts`. It doesn't matter what code follows a `halt`, since it won't be evaluated.

Individual processes can be killed by calling `kill`. If given no arguments, this operator kills the current process. In this case, `kill` is like a wait expression which neglects to store the current process. If `kill` is given arguments, they become the arguments to a `delete` on the list of processes. In the current code, there is not much one can say in a `kill` expression, because processes do not have many properties to refer to. However, a more elaborate system would associate more information with processes—time stamps, owners, and so on. The default process can't be killed, because it isn't kept in the list `*procs*`.

21.3 The Less-than-Rapid Prototype

Processes simulated with continuations are not going to be nearly as efficient as real operating system processes. What's the use, then, of programs like the one in this chapter?

Such programs are useful in the same way that sketches are. In *exploratory programming* or *rapid prototyping,* the program is not an end in itself so much as a vehicle for working out one's ideas. In many other fields, something which serves this purpose is called a sketch. An architect could, in principle, design an entire building in his head. However, most architects seem to think better with pencils in their hands: the design of a building is usually worked out in a series of preparatory sketches.

Rapid prototyping is sketching software. Like an architect's first sketches, software prototypes tend to be drawn with a few sweeping strokes. Considerations of cost and efficiency are ignored in an initial push to develop an idea to the full. The result, at this stage, is likely to be an unbuildable building or a hopelessly inefficient piece of software. But the sketches are valuable all the same, because

1. They convey information briefly.

2. They offer a chance to experiment.

The program described in this chapter is, like those in succeeding chapters, a sketch. It suggests the outlines of multiprocessing in a few, broad strokes. And though it would not be efficient enough for use in production software, it could be quite useful for experimenting with other aspects of multiple processes, like scheduling algorithms.

Chapters 22–24 present other applications of continuations. None of them is efficient enough for use in production software. Because Lisp and rapid prototyping evolved together, Lisp includes a lot of features specifically intended for prototypes: inefficient but convenient features like property lists, keyword parameters, and, for that matter, lists. Continuations probably belong in this category. They save more state than a program is likely to need. So our continuation-based implementation of Prolog, for example, is a good way to understand the language, but an inefficient way to implement it.

This book is concerned more with the kinds of abstractions one can build in Lisp than with efficiency issues. It's important to realize, though, that Lisp is a language for writing production software as well as a language for writing prototypes. If Lisp has a reputation for slowness, it is largely because so many programmers stop with the prototype. It is easy to write fast programs in Lisp. Unfortunately, it is *very* easy to write slow ones. The initial version of a Lisp program can be like a diamond: small, clear, and very expensive. There may be a great temptation to leave it that way.

In other languages, once you succeed in the arduous task of getting your program to work, it may already be acceptably efficient. If you tile a floor with tiles the size of your thumbnail, you don't waste many. Someone used to developing software on this principle may find it difficult to overcome the idea that when a program works, it's finished. "In Lisp you can write programs in no time at all," he may think, "but boy, are they slow." In fact, neither is the case. You can get fast programs, but you have to work for them. In this respect, using Lisp is like living in a rich country instead of a poor one: it may seem unfortunate that one has to work to stay thin, but surely this is better than working to stay alive, and being thin as a matter of course.

In less abstract languages, you work for functionality. In Lisp you work for speed. Fortunately, working for speed is easier: most programs only have a few critical sections in which speed matters.

22

Nondeterminism

Programming languages save us from being swamped by a mass of detail. Lisp is a good language because it handles so many details itself, enabling programmers to make the most of their limited tolerance for complexity. This chapter describes how macros can make Lisp handle another important class of details: the details of transforming a nondeterministic algorithm into a deterministic one.

This chapter is divided into five parts. The first explains what nondeterminism is. The second describes a Scheme implementation of nondeterministic *choose* and *fail* which uses continuations. The third part presents Common Lisp versions of *choose* and *fail* which build upon the continuation-passing macros of Chapter 20. The fourth part shows how the cut operator can be understood independently of Prolog. The final part suggests refinements of the original nondeterministic operators.

The nondeterministic choice operators defined in this chapter will be used to write an ATN compiler in Chapter 23 and an embedded Prolog in Chapter 24.

22.1 The Concept

A nondeterministic algorithm is one which relies on a certain sort of supernatural foresight. Why talk about such algorithms when we don't have access to computers with supernatural powers? Because a nondeterministic algorithm can be *simulated* by a deterministic one. For purely functional programs—that is, those with no side-effects—simulating nondeterminism is particularly straightforward. In purely functional programs, nondeterminism can be implemented by search with backtracking.

This chapter shows how to simulate nondeterminism in functional programs. If we have a simulator for nondeterminism, we can expect it to produce results whenever a truly nondeterministic machine would. In many cases, writing a program which depends on supernatural insight to solve a problem is easier than writing one which doesn't, so such a simulator would be a good thing to have.

In this section we will define the class of powers that nondeterminism allows us; the next section demonstrates their utility in some sample programs. The examples in these first two sections are written in Scheme. (Some differences between Scheme and Common Lisp are summarized on page 259.)

A nondeterministic algorithm differs from a deterministic one because it can use the two special operators *choose* and *fail*. *Choose* is a function which takes a finite set and returns one element. To explain how *choose* chooses, we must first introduce the concept of a computational *future*.

Here we will represent *choose* as a function `choose` which takes a list and returns one element. For each element, there is a set of futures the computation could have if that element were chosen. In the following expression

```
(let ((x (choose '(1 2 3))))
  (if (odd? x)
      (+ x 1)
      x))
```

there are three possible futures for the computation when it reaches the point of the `choose`:

1. If `choose` returns `1`, the computation will go through the then-clause of the `if`, and will return `2`.

2. If `choose` returns `2`, the computation will go through the else-clause of the `if`, and will return `2`.

3. If `choose` returns `3`, the computation will go through the then-clause of the `if`, and will return `4`.

In this case, we know exactly what the future of the computation will be as soon as we see what `choose` returns. In the general case, each choice is associated with a *set* of possible futures, because within some futures there could be additional `choose`s. For example, with

```
(let ((x (choose '(2 3))))
  (if (odd? x)
      (choose '(a b))
      x))
```

there are two sets of futures at the time of the first `choose`:

1. If `choose` returns `2`, the computation will go through the else-clause of the `if`, and will return `2`.
2. If `choose` returns `3`, the computation will go through the then-clause of the `if`. At this point, the path of the computation splits into two possible futures, one in which `a` is returned, and one in which `b` is.

The first set has one future and the second set has two, so the computation has three possible futures.

The point to remember is, if *choose* is given a choice of several alternatives, each one is associated with a set of possible futures. Which choice will it return? We can assume that *choose* works as follows:

1. It will only return a choice for which some future does not contain a call to *fail*.
2. A *choose* over zero alternatives is equivalent to a *fail*.

So, for example, in

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (fail)
      x))
```

each of the possible choices has exactly one future. Since the future for a choice of `1` contains a call to `fail`, only `2` can be chosen. So the expression as a whole is deterministic: it always returns `2`.

However, the following expression is not deterministic:

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (let ((y (choose '(a b))))
        (if (eq? y 'a)
            (fail)
            y))
      x))
```

At the first `choose`, there are two possible futures for a choice of `1`, and one for a choice of `2`. Within the former, though, the future is really deterministic, because a choice of `a` would result in a call to `fail`. So the expression as a whole could return either `b` or `2`.

Finally, there is only one possible value for the expression

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (choose '())
      x))
```

because if `1` is chosen, the future goes through a `choose` with no choices. This example is thus equivalent to the last but one.

It may not be clear yet from the preceding examples, but we have just got ourselves an abstraction of astounding power. In nondeterministic algorithms we are allowed to say "choose an element such that nothing we do later will result in a call to *fail*." For example, this is a perfectly legitimate nondeterministic algorithm for discovering whether you have a known ancestor called Igor:

Function Ig(n)
  if name(n) = 'Igor'
    then return n
  else if parents(n)
    then return Ig(*choose*(parents(n)))
  else *fail*

The *fail* operator is used to influence the value returned by *choose*. If we ever encounter a *fail*, *choose* would have chosen incorrectly. By definition *choose* guesses correctly. So if we want to guarantee that the computation will never pursue a certain path, all we need do is put a *fail* somewhere in it, and that path will never be followed. Thus, as it works recursively through generations of ancestors, the function Ig is able to choose at each step a path which leads to an Igor—to guess whether to follow the mother's or father's line.

It is as if a program can specify that *choose* pick some element from a set of alternatives, use the value returned by *choose* for as long as it wants, and then retroactively decide, by using *fail* as a veto, what it wants *choose* to have picked. And, presto, it turns out that that's just what *choose* did return. Hence the model in which *choose* has foresight.

In reality *choose* cannot have supernatural powers. Any implementation of *choose* must simulate correct guessing by backtracking when it discovers mistakes, like a rat finding its way through a maze. But all this backtracking can be done beneath the surface. Once you have some form of *choose* and *fail*, you get to write algorithms like the one above, as if it really were possible to guess what ancestor to follow. By using *choose* it is possible to write an algorithm to search some problem space just by writing an algorithm to traverse it.

```
(define (descent n1 n2)
  (if (eq? n1 n2)
      (list n2)
      (let ((p (try-paths (kids n1) n2)))
        (if p (cons n1 p) #f))))

(define (try-paths ns n2)
  (if (null? ns)
      #f
      (or (descent (car ns) n2)
          (try-paths (cdr ns) n2))))
```

Figure 22.1: Deterministic tree search.

```
(define (descent n1 n2)
  (cond ((eq? n1 n2) (list n2))
        ((null? (kids n1)) (fail))
        (else (cons n1 (descent (choose (kids n1)) n2)))))
```

Figure 22.2: Nondeterministic tree search.

22.2 Search

Many classic problems can be formulated as search problems, and for such problems nondeterminism often turns out to be a useful abstraction. Suppose `nodes` is bound to a list of nodes in a tree, and `(kids` *n*`)` is a function which returns the descendants of node *n*, or `#f` if there are none. We want to write a function `(descent` n_1 n_2`)` which returns a list of nodes on some path from n_1 to its descendant n_2, if there is one. Figure 22.1 shows a deterministic version of this function.

Nondeterminism allows the programmer to ignore the details of finding a path. It's possible simply to tell `choose` to find a node *n* such that there is a path from *n* to our destination. Using nondeterminism we can write the simpler version of `descent` shown in Figure 22.2.

The version shown in Figure 22.2 does not explicitly search for a node on the right path. It is written on the assumption that `choose` has chosen an `n` with the desired properties. If we are used to looking at deterministic programs, we may not perceive that `choose` has to work as if it could *guess* what `n` would make it

```
(define (two-numbers)
  (list (choose '(0 1 2 3 4 5))
        (choose '(0 1 2 3 4 5))))

(define (parlor-trick sum)
  (let ((nums (two-numbers)))
    (if (= (apply + nums) sum)
        `(the sum of ,@nums)
        (fail))))
```

Figure 22.3: Choice in a subroutine.

through the computation which follows without failing.

Perhaps a more convincing example of the power of *choose* is its ability to guess what will happen even in calling functions. Figure 22.3 contains a pair of functions to guess two numbers which sum to a number given by the caller. The first function, `two-numbers`, nondeterministically chooses two numbers and returns them in a list. When we call `parlor-trick`, it calls `two-numbers` for a list of two integers. Note that, in making its choice, `two-numbers` doesn't have access to the number entered by the user.

If the two numbers guessed by `choose` don't sum to the number entered by the user, the computation fails. We can rely on `choose` having avoided computational paths which fail, if there are any which don't. Thus we can assume that if the caller gives a number in the right range, `choose` will have guessed right, as indeed it does:[1]

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

In simple searches, the built-in Common Lisp function `find-if` would do just as well. Where is the advantage of nondeterministic choice? Why not just iterate through the list of alternatives in search of the element with the desired properties? The crucial difference between *choose* and conventional iteration is that its extent with respect to *fails* is unbounded. Nondeterministic *choose* can see arbitrarily far into the future; if something is going to happen at any point in the future which would have invalidated some guess *choose* might make, we can assume that *choose* knows to avoid guessing it. As we saw in `parlor-trick`,

[1] Since the order of argument evaluation is unspecified in Scheme (as opposed to Common Lisp, which specifies left-to-right), this call might also return `(THE SUM OF 5 2)`.

the *fail* operator works even after we return from the function in which the *choose* occurs.

This kind of failure happens in the search done by Prolog, for example. Nondeterminism is useful in Prolog because one of the central features of this language is its ability to return answers to a query one at a time. By following this course instead of returning all the valid answers at once, Prolog can handle recursive rules which would otherwise yield infinitely large sets of answers.

The initial reaction to `descent` may be like the initial reaction to a merge sort: where does the work get done? As in a merge sort, the work gets done implicitly, but it does get done. Section 22.3 describes an implementation of *choose* in which all the code examples presented so far are real running programs.

These examples show the value of nondeterminism as an abstraction. The best programming language abstractions save not just typing, but thought. In automata theory, some proofs are difficult even to conceive of without relying on nondeterminism. A language which allows nondeterminism may give programmers a similar advantage.

22.3 Scheme Implementation

This section explains how to use continuations to simulate nondeterminism. Figure 22.4 contains Scheme implementations of *choose* and *fail*. Beneath the surface, `choose` and `fail` simulate nondeterminism by backtracking. A backtracking search program must somehow store enough information to pursue other alternatives if the chosen one fails. This information is stored in the form of continuations on the global list `*paths*`.

The function `choose` is passed a list of alternatives in `choices`. If `choices` is empty, then `choose` calls `fail`, which sends the computation back to the previous `choose`. If `choices` is `(`*first* `.` *rest*`)`, `choose` first pushes onto `*paths*` a continuation in which `choose` is called on *rest*, then returns *first*.

The function `fail` is simpler: it just pops a continuation off `*paths*` and calls it. If there aren't any saved paths left, then `fail` returns the symbol `@`. However, it won't do simply to return it as a function ordinarily returns values, or it will be returned as the value of the most recent `choose`. What we really want to do is return `@` right to the toplevel. We do this by binding `cc` to the continuation where `fail` is defined, which presumably is the toplevel. By calling `cc`, `fail` can return straight there.

The implementation in Figure 22.4 treats `*paths*` as a stack, always failing back to the most recent choice point. This strategy, known as *chronological backtracking,* results in depth-first search of the problem space. The word "nondeterminism" is often used as if it were synonymous with the depth-first imple-

```
(define *paths* ())
(define failsym '@)

(define (choose choices)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (set! *paths*
                (cons (lambda ()
                        (cc (choose (cdr choices))))
                      *paths*))
        (car choices)))))

(define fail)

(call-with-current-continuation
  (lambda (cc)
    (set! fail
          (lambda ()
            (if (null? *paths*)
                (cc failsym)
                (let ((p1 (car *paths*)))
                  (set! *paths* (cdr *paths*))
                  (p1)))))))
```

Figure 22.4: Scheme implementation of *choose* and *fail*.

mentation. Floyd's classic paper on nondeterministic algorithms uses the term in this sense, and this is also the kind of nondeterminism we find in nondeterministic parsers and in Prolog. However, it should be noted that the implementation given in Figure 22.4 is not the only possible implementation, nor even a correct one. In principle, *choose* ought to be able to choose an object which meets any computable specification. But a program which used these versions of `choose` and `fail` to search a graph might not terminate, if the graph contained cycles.

In practice, nondeterminism usually means using a depth-first implementation equivalent to the one in Figure 22.4, and leaving it to the user to avoid loops in the search space. However, for readers who are interested, the last section in this chapter describes how to implement true *choose* and *fail*.

22.4 Common Lisp Implementation

This section describes how to write a form of *choose* and *fail* in Common Lisp. As the previous section showed, `call/cc` makes it easy to simulate nondeterminism in Scheme. Continuations provide the direct embodiment of our theoretical concept of a computational future. In Common Lisp, we can use instead the continuation-passing macros of Chapter 20. With these macros we will be able to provide a form of *choose* slightly uglier than the Scheme version presented in the previous section, but equivalent in practice.

Figure 22.5 contains a Common Lisp implementation of *fail*, and two versions of *choose*. The syntax of a Common Lisp `choose` is slightly different from the Scheme version. The Scheme `choose` took one argument: a list of choices from which to select a value. The Common Lisp version has the syntax of a `progn`. It can be followed by any number of expressions, from which it chooses one to evaluate:

```
> (defun do2 (x)
    (choose (+ x 2) (* x 2) (expt x 2)))
DO2
> (do2 3)
5
> (fail)
6
```

At the toplevel, we see more clearly the backtracking which underlies nondeterministic search. The variable `*paths*` is used to store paths which have not yet been followed. When the computation reaches a `choose` expression with several alternatives, the first alternative is evaluated, and the remaining choices are stored on `*paths*`. If the program later on encounters a `fail`, the last stored choice will be popped off `*paths*` and restarted. When there are no more paths left to restart, `fail` returns a special value:

```
> (fail)
9
> (fail)
@
```

In Figure 22.5 the constant `failsym`, which represents failure, is defined to be the symbol `@`. If you wanted to be able to have `@` as an ordinary return value, you could make `failsym` a gensym instead.

The other nondeterministic choice operator, `choose-bind`, has a slightly different form. It should be given a symbol, a list of choices, and a body of code. It will do a *choose* on the list of choices, bind the symbol to the value chosen, and evaluate the body of code:

```
(defparameter *paths* nil)
(defconstant failsym '@)

(defmacro choose (&rest choices)
  (if choices
      `(progn
         ,@(mapcar #'(lambda (c)
                       `(push #'(lambda () ,c) *paths*))
                   (reverse (cdr choices)))
         ,(car choices))
      '(fail)))

(defmacro choose-bind (var choices &body body)
  `(cb #'(lambda (,var) ,@body) ,choices))

(defun cb (fn choices)
  (if choices
     (progn
       (if (cdr choices)
           (push #'(lambda () (cb fn (cdr choices)))
                 *paths*))
       (funcall fn (car choices)))
     (fail)))

(defun fail ()
  (if *paths*
      (funcall (pop *paths*))
      failsym))
```

Figure 22.5: Nondeterministic operators in Common Lisp.

```
> (choose-bind x '(marrakesh strasbourg vegas)
    (format nil "Let's go to ~A." x))
"Let's go to MARRAKESH."
> (fail)
"Let's go to STRASBOURG."
```

It is only for convenience that the Common Lisp implementation provides two choice operators. You could get the effect of choose from choose-bind by always translating

```
(choose (foo) (bar))
```

into

```
(choose-bind x '(1 2)
  (case x
    (1 (foo))
    (2 (bar))))
```

but programs are easier to read if we have a separate operator for this case.[2]

The Common Lisp choice operators store the bindings of relevant variables using closures and variable capture. As macros, `choose` and `choose-bind` get expanded within the lexical environment of the containing expressions. Notice that what they push onto `*paths*` is a closure over the choice to be saved, locking in all the bindings of the lexical variables referred to within it. For example, in the expression

```
(let ((x 2))
  (choose
    (+ x 1)
    (+ x 100)))
```

the value of `x` will be needed when the saved choices are restarted. This is why `choose` is written to wrap its arguments in lambda-expressions. The expression above gets macroexpanded into:

```
(let ((x 2))
  (progn
    (push #'(lambda () (+ x 100))
          *paths*)
    (+ x 1)))
```

The object which gets stored on `*paths*` is a closure containing a pointer to `x`. It is the need to preserve variables in closures which dictates the difference between the syntax of the Scheme and Common Lisp choice operators.

If we use `choose` and `fail` together with the continuation-passing macros of Chapter 20, a pointer to our continuation variable `*cont*` will get saved as well. By defining functions with `=defun`, calling them with `=bind`, and having them return values with `=values`, we will be able to use nondeterminism in any Common Lisp program.

With these macros, we can successfully run the example in which the nondeterministic choice occurs in a subroutine. Figure 22.6 shows the Common Lisp version of `parlor-trick`, which works as it did in Scheme:

[2] If desired, the exported interface to this code could consist of just a single operator, because `(fail)` is equivalent to `(choose)`.

```
(=defun two-numbers ()
  (choose-bind n1 '(0 1 2 3 4 5)
    (choose-bind n2 '(0 1 2 3 4 5)
      (=values n1 n2))))

(=defun parlor-trick (sum)
  (=bind (n1 n2) (two-numbers)
    (if (= (+ n1 n2) sum)
        `(the sum of ,n1 ,n2)
        (fail))))
```

Figure 22.6: Common Lisp choice in a subroutine.

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

This works because the expression

```
(=values n1 n2)
```

gets macroexpanded into

```
(funcall *cont* n1 n2)
```

within the `choose-bind`s. Each `choose-bind` is in turn macroexpanded into a closure, which keeps pointers to all the variables referred to in the body, including `*cont*`.

The restrictions on the use of `choose`, `choose-bind`, and `fail` are the same as the restrictions given in Figure 20.5 for code which uses the continuation-passing macros. Where a choice expression occurs, it must be the last thing to be evaluated. Thus if we want to make sequential choices, in Common Lisp the choices have to be nested:

```
> (choose-bind first-name '(henry william)
    (choose-bind last-name '(james higgins)
      (=values (list first-name last-name))))
(HENRY JAMES)
> (fail)
(HENRY HIGGINS)
> (fail)
(WILLIAM JAMES)
```

which will, as usual, result in depth-first search.

The operators defined in Chapter 20 claimed the right to be the last expressions evaluated. This right is now preempted by the new layer of macros; an `=values` expression should appear within a `choose` expression, and not vice versa. That is,

```
(choose (=values 1) (=values 2))
```

will work, but

```
(=values (choose 1 2))                                    ; wrong
```

will not. (In the latter case, the expansion of the `choose` would be unable to capture the instance of `*cont*` in the expansion of the `=values`.)

As long as we respect the restrictions outlined here and in Figure 20.5, nondeterministic choice in Common Lisp will now work as it does in Scheme. Figure 22.7 shows a Common Lisp version of the nondeterministic tree search program given in Figure 22.2. The Common Lisp `descent` is a direct translation, though it comes out slightly longer and uglier.

We now have Common Lisp utilities which make it possible to do nondeterministic search without explicit backtracking. Having taken trouble to write this code, we can reap the benefits by writing in very few lines programs which would otherwise be large and messy. By building another layer of macros on top of those presented here, we will be able to write an ATN compiler in one page of code (Chapter 23), and a sketch of Prolog in two (Chapter 24).

Common Lisp programs which use `choose` should be compiled with tail-recursion optimization—not just to make them faster, but to avoid running out of stack space. Programs which "return" values by calling continuation functions never actually return until the final `fail`. Without the optimization of tail-calls, ○ the stack would just grow and grow.

22.5 Cuts

This section shows how to use cuts in Scheme programs which do nondeterministic choice. Though the word *cut* comes from Prolog, the concept belongs to nondeterminism generally. You might want to use cuts in any program that made nondeterministic choices.

Cuts are easier to understand when considered independently of Prolog. Let's imagine a real-life example. Suppose that the manufacturer of Chocoblob candies decides to run a promotion. A small number of boxes of Chocoblobs will also contain tokens entitling the recipient to valuable prizes. To ensure fairness, no two of the winning boxes are sent to the same city.

```
> (=defun descent (n1 n2)
    (cond ((eq n1 n2) (=values (list n2)))
          ((kids n1) (choose-bind n (kids n1)
                       (=bind (p) (descent n n2)
                          (=values (cons n1 p)))))
          (t (fail))))
DESCENT
> (defun kids (n)
    (case n
      (a '(b c))
      (b '(d e))
      (c '(d f))
      (f '(g))))
KIDS
> (descent 'a 'g)
(A C F G)
> (fail)
@
> (descent 'a 'd)
(A B D)
> (fail)
(A C D)
> (fail)
@
> (descent 'a 'h)
@
```

Figure 22.7: Nondeterministic search in Common Lisp

After the promotion has begun, it emerges that the tokens are small enough to be swallowed by children. Hounded by visions of lawsuits, Chocoblob lawyers begin a frantic search for all the special boxes. Within each city, there are multiple stores that sell Chocoblobs; within each store, there are multiple boxes. But the lawyers may not have to open every box: once they find a coin-containing box in a given city, they do not have to search any of the other boxes in that city, because each city has at most one special box. To realize this is to do a cut.

What's *cut* is a portion of the search tree. For Chocoblobs, the search tree exists physically: the root node is at the company's head office; the children of this node are the cities where the special boxes were sent; the children of those nodes are the stores in each city; and the children of each store represent the boxes in

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (newline)
    (let* ((store (choose '(1 2)))
           (box (choose '(1 2))))
      (let ((triple (list city store box)))
        (display triple)
        (if (coin? triple)
            (display 'c))
        (fail)))))

(define (coin? x)
  (member x '((la 1 2) (ny 1 1) (bos 2 2))))
```

Figure 22.8: Exhaustive Chocoblob search.

that store. When the lawyers searching this tree find one of the boxes containing a coin, they can prune off all the unexplored branches descending from the city they're in now.

Cuts actually take two operations: you can do a cut when you know that part of the search tree is useless, but first you have to *mark* the tree at the point where it can be cut. In the Chocoblob example, common sense tells us that the tree is marked as we enter each city. It's hard to describe in abstract terms what a Prolog cut does, because the marks are implicit. With an explicit mark operator, the effect of a cut will be more easily understood.

Figure 22.8 shows a program that nondeterministically searches a smaller version of the Chocoblob tree. As each box is opened, the program displays a list of (*city store box*). If the box contains a coin, a `c` is printed after it:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C(LA 2 1)(LA 2 2)
(NY 1 1)C(NY 1 2)(NY 2 1)(NY 2 2)
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
@
```

To implement the optimized search technique discovered by the Chocoblob lawyers, we need two new operators: `mark` and `cut`. Figure 22.9 shows one way to define them. Whereas nondeterminism itself can be understood independently of any particular implementation, pruning the search tree is an optimization technique, and depends very much on how `choose` is implemented. The `mark` and

```
(define (mark) (set! *paths* (cons fail *paths*)))

(define (cut)
  (cond ((null? *paths*))
        ((equal? (car *paths*) fail)
         (set! *paths* (cdr *paths*)))
        (else
         (set! *paths* (cdr *paths*))
         (cut))))
```

Figure 22.9: Marking and pruning search trees.

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (mark)                                                    ;
    (newline)
    (let* ((store (choose '(1 2)))
           (box (choose '(1 2))))
      (let ((triple (list city store box)))
        (display triple)
        (if (coin? triple)
            (begin (cut) (display 'c)))                       ;
        (fail)))))
```

Figure 22.10: Pruned Chocoblob search.

`cut` defined in Figure 22.9 are suitable for use with the depth-first implementation of `choose` (Figure 22.4).

The general idea is for `mark` to store markers in `*paths*`, the list of unexplored choice-points. Calling `cut` pops `*paths*` all the way down to the most recent marker. What should we use as a marker? We could use e.g. the symbol `m`, but that would require us to rewrite `fail` to ignore the `m`s when it encountered them. Fortunately, since functions are data objects too, there is at least one marker that will allow us to use `fail` as is: the function `fail` itself. Then if `fail` happens on a marker, it will just call itself.

Figure 22.10 shows how these operators would be used to prune the search tree in the Chocoblob case. (Changed lines are indicated by semicolons.) We call `mark` upon choosing a city. At this point, `*paths*` contains one continuation,

Figure 22.11: A directed graph with a loop.

representing the search of the remaining cities.

If we find a box with a coin in it, we call `cut`, which sets `*paths*` back to the value it had at the time of the `mark`. The effects of the cut are not visible until the next call to `fail`. But when it comes, after the `display`, the next `fail` sends the search all the way up to the topmost `choose`, even if there would otherwise have been live choice-points lower in the search tree. The upshot is, as soon as we find a box with a coin in it, we resume the search at the next city:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C
(NY 1 1)C
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
@
```

In this case, we open seven boxes instead of twelve.

22.6 True Nondeterminism

A deterministic graph-searching program would have to take explicit steps to avoid getting caught in a circular path. Figure 22.11 shows a directed graph containing a loop. A program searching for a path from node `a` to node `e` risks getting caught in the circular path ⟨`a`, `b`, `c`⟩. Unless a deterministic searcher used randomization, breadth-first search, or checked explicitly for circular paths, the search might never terminate. The implementation of `path` shown in Figure 22.12 avoids circular paths by searching breadth-first.

In principle, nondeterminism should save us the trouble of even considering circular paths. The depth-first implementation of *choose* and *fail* given in Section 22.3 is vulnerable to the problem of circular paths, but if we were being picky, we would expect nondeterministic *choose* to be able to select an object

```
(define (path node1 node2)
  (bf-path node2 (list (list node1))))

(define (bf-path dest queue)
  (if (null? queue)
      '@
      (let* ((path (car queue))
             (node (car path)))
        (if (eq? node dest)
            (cdr (reverse path))
            (bf-path dest
                     (append (cdr queue)
                             (map (lambda (n)
                                    (cons n path))
                                  (neighbors node))))))))
```

Figure 22.12: Deterministic search.

```
(define (path node1 node2)
  (cond ((null? (neighbors node1)) (fail))
        ((memq node2 (neighbors node1)) (list node2))
        (else (let ((n (true-choose (neighbors node1))))
                (cons n (path n node2))))))
```

Figure 22.13: Nondeterministic search.

which meets any computable specification, and this case is no exception. Using a correct *choose*, we should be able to write the shorter and clearer version of `path` shown in Figure 22.13.

This section shows how to implement versions *choose* and *fail* which are safe even from circular paths. Figure 22.14 contains a Scheme implementation of true nondeterministic *choose* and *fail*. Programs which use these versions of *choose* and *fail* should find solutions whenever the equivalent nondeterministic algorithms would, subject to hardware limitations.

The implementation of `true-choose` defined in Figure 22.14 works by treating the list of stored paths as a queue. Programs using `true-choose` will search their state-space breadth-first. When the program reaches a choice-point, continuations to follow each choice are appended to the end of the list of stored paths.

```
(define *paths* ())
(define failsym '@)

(define (true-choose choices)
  (call-with-current-continuation
    (lambda (cc)
      (set! *paths* (append *paths*
                            (map (lambda (choice)
                                   (lambda () (cc choice)))
                                 choices)))
      (fail))))

(define fail)

(call-with-current-continuation
  (lambda (cc)
    (set! fail
          (lambda ()
            (if (null? *paths*)
                (cc failsym)
                (let ((p1 (car *paths*)))
                  (set! *paths* (cdr *paths*))
                  (p1)))))))
```

Figure 22.14: Correct *choose* in Scheme.

(Scheme's `map` returns the same values as Common Lisp's `mapcar`.) After this there is a call to `fail`, which is unchanged.

This version of *choose* would allow the implementation of `path` defined in Figure 22.13 to find a path—indeed, the shortest path—from `a` to `e` in the graph displayed in Figure 22.11.

Although for the sake of completeness this chapter has provided correct versions of *choose* and *fail*, the original implementations will usually suffice. The value of a programming language abstraction is not diminished just because its implementation isn't formally correct. In some languages we act as if we had access to all the integers, even though the largest one may be only 32767. As long as we know how far we can push the illusion, there is little danger to it—little enough, at least, to make the abstraction a bargain. The conciseness of the programs presented in the next two chapters is due largely to their use of nondeterministic *choose* and *fail*.

23

Parsing with ATNs

This chapter shows how to write a nondeterministic parser as an embedded language. The first part explains what ATN parsers are, and how they represent grammar rules. The second part presents an ATN compiler which uses the nondeterministic operators defined in the previous chapter. The final sections present a small ATN grammar, and show it in action parsing sample input.

23.1 Background

Augmented Transition Networks, or ATNs, are a form of parser described by Bill Woods in 1970. Since then they have become a widely used formalism for ○ parsing natural language. In an hour you can write an ATN grammar which parses interesting English sentences. For this reason, people are often held in a sort of spell when they first encounter them.

In the 1970s, some people thought that ATNs might one day be components of truly intelligent-seeming programs. Though few hold this position today, ATNs have found a niche. They aren't as good as you are at parsing English, but they can still parse an impressive variety of sentences.

ATNs are useful if you observe the following four restrictions:

1. Use them in a semantically limited domain—in a front-end to a particular database, for example.

2. Don't feed them very difficult input. Among other things, don't expect them to understand wildly ungrammatical sentences the way people can.

3. Only use them for English, or other languages in which word order determines grammatical structure. ATNs would not be useful in parsing inflected languages like Latin.

4. Don't expect them to work all the time. Use them in applications where it's helpful if they work ninety percent of the time, not those where it's critical that they work a hundred percent of the time.

Within these limits there are plenty of useful applications. The canonical example is as the front-end of a database. If you attach an ATN-driven interface to such a system, then instead of making a formal query, users can ask questions in a constrained form of English.

23.2 The Formalism

To understand what ATNs do, we should recall their full name: augmented transition networks. A transition network is a set of nodes joined together by directed arcs—essentially, a flow-chart. One node is designated the start node, and some other nodes are designated terminal nodes. Conditions are attached to each arc, which have to be met before the arc can be followed. There will be an input sentence, with a pointer to the current word. Following some arcs will cause the pointer to be advanced. To parse a sentence on a transition network is to find a path from the start node to some terminal node, along which all the conditions can be met.

ATNs add two features to this model:

1. ATNs have registers—named slots for storing away information as the parse proceeds. As well as performing tests, arcs can modify the contents of the registers.

2. ATNs are recursive. Arcs may require that, in order to follow them, the parse must successfully make it through some sub-network.

Terminal nodes use the information which has accumulated in the registers to build list structures, which they return in much the same way that functions return values. In fact, with the exception of being nondeterministic, ATNs behave a lot like a functional programming language.

The ATN defined in Figure 23.1 is nearly the simplest possible. It parses noun-verb sentences of the form "Spot runs." The network representation of this ATN is shown in Figure 23.2.

What does this ATN do when given the input `(spot runs)`? The first node has one outgoing arc, a cat, or category arc, leading to node `s2`. It says, effectively,

```
(defnode s
  (cat noun s2
    (setr subj *)))

(defnode s2
  (cat verb s3
    (setr v *)))

(defnode s3
  (up `(sentence
         (subject ,(getr subj))
         (verb ,(getr v)))))
```

Figure 23.1: A very small ATN.

Figure 23.2: Graph of a small ATN.

you can follow me if the current word is a noun, and if you do, you must store the current word (indicated by `*`) in the `subj` register. So we leave this node with `spot` stored in the `subj` register.

There is always a pointer to the current word. Initially it points to the first word in the sentence. When cat arcs are followed, this pointer is moved forward one. So when we get to node `s2`, the current word is the second, `runs`. The second arc is just like the first, except that it is looking for a verb. It finds `runs`, stores it in register `v`, and proceeds to `s3`.

The final node, `s3`, has only a pop, or terminal, arc. (Nodes with pop arcs have dashed borders.) Because we arrive at the pop arc just as we run out of input, we have a successful parse. The pop arc returns the backquoted expression within it:

```
(sentence (subject spot)
          (verb runs))
```

An ATN corresponds to the grammar of the language it is designed to parse. A decent-sized ATN for parsing English will have a main network for parsing sentences, and sub-networks for parsing noun-phrases, prepositional phrases, modifier groups, and so on. The need for recursion is obvious when we consider that noun-phrases may contain prepositional phrases which may contain noun-phrases, *ad infinitum,* as in

> "the key on the table in the hall of the house on the hill"

23.3 Nondeterminism

Although we didn't see it in this small example, ATNs are nondeterministic. A node can have several outgoing arcs, more than one of which could be followed with a given input. For example, a reasonably good ATN should be able to parse both imperative and declarative sentences. Thus the first node could have outgoing cat arcs for both nouns (in statements) and verbs (in commands).

What if the first word of the sentence is "time," which is both a noun and a verb? How does the parser know which arc to follow? When ATNs are described as nondeterministic, it means that users can assume that the parser will *correctly guess* which arc to follow. If some arcs lead only to failed parses, they won't be followed.

In reality the parser cannot look into the future. It simulates correct guessing by backtracking when it runs out of arcs, or input. But all the machinery of backtracking is inserted automatically into the code generated by the ATN compiler. We can write ATNs as if the parser really could guess which arcs to follow.

Like many (perhaps most) programs which use nondeterminism, ATNs use the depth-first implementation. Experience parsing English quickly teaches one that any given sentence has a slew of legal parsings, most of them junk. On a conventional single-processor machine, one is better off trying to get good parses quickly. Instead of getting all the parses at once, we get just the most likely. If it has a reasonable interpretation, then we have saved the effort of finding other parses; if not, we can call *fail* to get more.

To control the order in which parses are generated, the programmer needs to have some way of controlling the order in which *choose* tries alternatives. The depth-first implementation isn't the only way of controlling the order of the search. Any implementation except a randomizing one imposes some kind of order. However, ATNs, like Prolog, have the depth-first implementation conceptually built-in. In an ATN, the arcs leaving a node are tried in the order in which they were defined. This convention allows the programmer to order arcs by priority.

23.4 An ATN Compiler

Ordinarily, an ATN-based parser needs three components: the ATN itself, an interpreter for traversing it, and a dictionary which can tell it, for example, that "runs" is a verb. Dictionaries are a separate topic—here we will use a rudimentary handmade one. Nor will we need to deal with a network interpreter, because we will translate the ATN directly into Lisp code. The program described here is called an ATN compiler because it transforms a whole ATN into code. Nodes are transformed into functions, and arcs become blocks of code within them.

Chapter 6 introduced the use of functions as a form of representation. This practice usually makes programs faster. Here it means that there will be no overhead of interpreting the network at runtime. The disadvantage is that there is less to inspect when something goes wrong, especially if you're using a Common Lisp implementation which doesn't provide `function-lambda-expression`.

Figure 23.3 contains all the code for transforming ATN nodes into Lisp code. The macro `defnode` is used to define nodes. It generates little code itself, just a `choose` over the expressions generated for each of the arcs. The two parameters of a node-function get the following values: `pos` is the current input pointer (an integer), and `regs` is the current set of registers (a list of assoc-lists).

The macro `defnode` defines a macro with the same name as the corresponding node. Node `s` will be defined as macro `s`. This convention enables arcs to know how to refer to their destination nodes—they just call the macro with that name. It also means that you shouldn't give nodes the names of existing functions or macros, or these will be redefined.

Debugging ATNs requires some sort of trace facility. Because nodes become functions, we don't have to write our own. We can use the built-in Lisp function `trace`. As mentioned on page 266, using `=defun` to define nodes means that we can trace parses going through node `mods` by saying `(trace =mods)`.

The arcs within the body of a node are simply macro calls, returning code which gets embedded in the node function being made by `defnode`. The parser uses nondeterminism at each node by executing a `choose` over the code representing each of the arcs leaving that node. A node with several outgoing arcs, say

```
(defnode foo
  <arc 1>
  <arc 2>)
```

gets translated into a function definition of the following form:

```
(=defun foo (pos regs)
  (choose
    <translation of arc 1>
    <translation of arc 2>))
```

```
(defmacro defnode (name &rest arcs)
  `(=defun ,name (pos regs) (choose ,@arcs)))

(defmacro down (sub next &rest cmds)
  `(=bind (* pos regs) (,sub pos (cons nil regs))
     (,next pos ,(compile-cmds cmds))))

(defmacro cat (cat next &rest cmds)
  `(if (= (length *sent*) pos)
       (fail)
       (let ((* (nth pos *sent*)))
         (if (member ',cat (types *))
             (,next (1+ pos) ,(compile-cmds cmds))
             (fail)))))

(defmacro jump (next &rest cmds)
  `(,next pos ,(compile-cmds cmds)))

(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds)))))

(defmacro up (expr)
  `(let ((* (nth pos *sent*)))
     (=values ,expr pos (cdr regs))))

(defmacro getr (key &optional (regs 'regs))
  `(let ((result (cdr (assoc ',key (car ,regs)))))
     (if (cdr result) result (car result))))

(defmacro set-register (key val regs)
  `(cons (cons (cons ,key ,val) (car ,regs))
         (cdr ,regs)))

(defmacro setr (key val regs)
  `(set-register ',key (list ,val) ,regs))

(defmacro pushr (key val regs)
  `(set-register ',key
                 (cons ,val (cdr (assoc ',key (car ,regs))))
                 ,regs))
```

Figure 23.3: Compilation of nodes and arcs.

```
(defnode s
  (down np s/subj
    (setr mood 'decl)
    (setr subj *))
  (cat v v
    (setr mood 'imp)
    (setr subj '(np (pron you)))
    (setr aux nil)
    (setr v *)))
```

is macroexpanded into:

```
(=defun s (pos regs)
  (choose
    (=bind (* pos regs) (np pos (cons nil regs))
      (s/subj pos
              (setr mood 'decl
                    (setr subj * regs))))
    (if (= (length *sent*) pos)
        (fail)
        (let ((* (nth pos *sent*)))
          (if (member 'v (types *))
              (v (1+ pos)
                 (setr mood 'imp
                       (setr subj '(np (pron you))
                             (setr aux nil
                                   (setr v * regs)))))
              (fail))))))
```

Figure 23.4: Macroexpansion of a node function.

Figure 23.4 shows the macroexpansion of the first node in the sample ATN of Figure 23.11. When called at runtime, node functions like `s` nondeterministically choose an arc to follow. The parameter `pos` will be the current position in the input sentence, and `regs` the current registers.

Cat arcs, as we saw in our original example, insist that the current word of input belong to a certain grammatical category. Within the body of a cat arc, the symbol `*` will be bound to the current word of input.

Push arcs, defined with `down`, require successful calls to sub-networks. They take two destination nodes, the sub-network destination `sub`, and the next node in the current network, `next`. Notice that whereas the code generated for a cat arc simply calls the next node in the network, the code generated for a push arc uses `=bind`. The push arc must successfully return from the sub-network before continuing on to the node which follows it. A clean set of registers (`nil`) gets consed onto the front of `regs` before they are passed to the sub-network. In the bodies of other types of arcs, the symbol `*` will be bound to the current word of input, but in push arcs it will be bound to the expression returned by the sub-network.

Jump arcs are like short-circuits. The parser skips right across to the destination node—no tests are required, and the input pointer isn't advanced.

The final type of arc is the pop arc, defined with `up`. Pop arcs are unusual in that they don't have a destination. Just as a Lisp `return` leads not to a subroutine but the calling function, a pop arc leads not to a new node but back to the "calling" push arc. The `=values` in a pop arc "returns" a value to the `=bind` in the most recent push arc. But, as Section 20.2 explained, what's happening is not a normal Lisp `return`: the body of the `=bind` has been wrapped up into a continuation and passed down as a parameter through any number of arcs, until the `=values` of the pop arc finally calls it on the "return" values.

Chapter 22 described two versions of nondeterministic *choose*: a fast `choose` (page 293) that wasn't guaranteed to terminate when there were loops in the search space, and a slower `true-choose` (page 304) which was safe from such loops. There can be cycles in an ATN, of course, but as long as at least one arc in each cycle advances the input pointer, the parser will eventually run off the end of the sentence. The problem arises with cycles which don't advance the input pointer. Here we have two alternatives:

1. Use the slower, correct nondeterministic choice operator (the depth-first version given on page 396).
2. Use the fast `choose`, and specify that it is an error to define networks containing cycles which could be traversed by following just jump arcs.

The code defined in Figure 23.3 takes the second approach.

The last four definitions in Figure 23.3 define the macros used to read and set registers within arc bodies. In this program, register sets are represented as assoc-lists. An ATN deals not with sets of registers, but sets of sets of registers. When the parser moves down to a sub-network, it gets a clean set of registers pushed on top of the existing ones. Thus the whole collection of registers, at any given time, is a list of assoc-lists.

The predefined register operators work on the current, or topmost, set of registers: `getr` reads a register; `setr` sets one; and `pushr` pushes a value into one. Both `getr` and `pushr` use the primitive register manipulation macro `set-register`.

Note that registers don't have to be declared. If `set-register` is sent a certain name, it will create a register with that name.

The register operators are all completely nondestructive. Cons, cons, cons, says `set-register`. This makes them slow and generates a lot of garbage, but, as explained on page 261, objects used in a part of a program where continuations are made should not be destructively modified. An object in one thread of control may be shared by another thread which is currently suspended. In this case, the registers found in one parse will share structure with the registers in many of the other parses. If speed became an issue, we could store registers in vectors instead of assoc-lists, and recycle used vectors into a common pool.

Push, cat, and jump arcs can all contain bodies of expressions. Ordinarily these will be just `setr`s. By calling `compile-cmds` on their bodies, the expansion functions of these arc types string a series of `setr`s into a single expression:

```
> (compile-cmds '((setr a b) (setr c d)))
(SETR A B (SETR C D REGS))
```

Each expression has the next expression inserted as its last argument, except the last, which gets `regs`. So a series of expressions in the body of an arc will be transformed into a single expression returning the new registers.

This approach allows users to insert arbitrary Lisp code into the bodies of arcs by wrapping it in a `progn`. For example:

```
> (compile-cmds '((setr a b)
                  (progn (princ "ek!"))
                  (setr c d)))
(SETR A B (PROGN (PRINC "ek!") (SETR C D REGS)))
```

Certain variables are left visible to code occurring in arc bodies. The sentence will be in the global `*sent*`. Two lexical variables will also be visible: `pos`, containing the current input pointer, and `regs`, containing the current registers. This is another example of intentional variable capture. If it were desirable to prevent the user from referring to these variables, they could be replaced with gensyms.

The macro `with-parses`, defined in Figure 23.5, gives us a way of invoking an ATN. It should be called with the name of a start node, an expression to be parsed, and a body of code describing what to do with the returned parses. The body of code within a `with-parses` expression will be evaluated once for each successful parse. Within the body, the symbol `parse` will be bound to the current parse. Superficially `with-parses` resembles operators like `dolist`, but underneath it uses backtracking search instead of simple iteration. A `with-parses` expression will return `@`, because that's what `fail` returns when it runs out of choices.

```
(defmacro with-parses (node sent &body body)
  (with-gensyms (pos regs)
    `(progn
       (setq *sent* ,sent)
       (setq *paths* nil)
       (=bind (parse ,pos ,regs) (,node 0 '(nil))
         (if (= ,pos (length *sent*))
             (progn ,@body (fail))
             (fail))))))
```

Figure 23.5: Toplevel macro.

Before going on to look at a more representative ATN, let's look at a parsing generated from the tiny ATN defined earlier. The ATN compiler (Figure 23.3) generates code which calls `types` to determine the grammatical roles of a word, so first we have to give it some definition:

```
(defun types (w)
  (cdr (assoc w '((spot noun) (runs verb)))))
```

Now we just call `with-parses` with the name of the start node as the first argument:

```
> (with-parses s '(spot runs)
    (format t "Parsing: ~A~%" parse))
Parsing: (SENTENCE (SUBJECT SPOT) (VERB RUNS))
@
```

23.5 A Sample ATN

Now that the whole ATN compiler has been described, we can go on to try out some parses using a sample network. In order to make an ATN parser handle a richer variety of sentences, you make the ATNs themselves more complicated, not the ATN compiler. The compiler presented here is a toy mainly in the sense that it's slow, not in the sense of having limited power.

The power (as distinct from speed) of a parser is in the grammar, and here limited space really will force us to use a toy version. Figures 23.8 through 23.11 define the ATN (or set of ATNs) represented in Figure 23.6. This network is just big enough to yield several parsings for the classic parser fodder "Time flies like an arrow."

Figure 23.6: Graph of a larger ATN.

```
(defun types (word)
  (case word
    ((do does did) '(aux v))
    ((time times) '(n v))
    ((fly flies) '(n v))
    ((like) '(v prep))
    ((liked likes) '(v))
    ((a an the) '(det))
    ((arrow arrows) '(n))
    ((i you he she him her it) '(pron))))
```

Figure 23.7: Nominal dictionary.

We need a slightly larger dictionary to parse more complex input. The function `types` (Figure 23.7) provides a dictionary of the most primitive sort. It defines a 22-word vocabulary, and associates each word with a list of one or more simple grammatical roles.

```
(defnode mods
  (cat n mods/n
    (setr mods *)))

(defnode mods/n
  (cat n mods/n
    (pushr mods *))
  (up `(n-group ,(getr mods))))
```

Figure 23.8: Sub-network for strings of modifiers.

The components of an ATN are themselves ATNs. The smallest ATN in our set is the one in Figure 23.8. It parses strings of modifiers, which in this case means just strings of nouns. The first node, `mods`, accepts a noun. The second node, `mods/n`, can either look for more nouns, or return a parsing.

Section 3.4 explained how writing programs in a functional style makes them easier to test:

1. In a functional program, components can be tested individually.
2. In Lisp, functions can be tested interactively, in the toplevel loop.

Together these two principles allow *interactive development:* when we write functional programs in Lisp, we can test each piece as we write it.

ATNs are so like functional programs—in this implementation, they macroexpand *into* functional programs—that the possibility of interactive development applies to them as well. We can test an ATN starting from any node, simply by giving its name as the first argument to `with-parses`:

```
> (with-parses mods '(time arrow)
    (format t "Parsing: ~A~%" parse))
Parsing: (N-GROUP (ARROW TIME))
@
```

The next two networks have to be discussed together, because they are mutually recursive. The network defined in Figure 23.9, which begins with the node `np`, is used to parse noun phrases. The network defined in Figure 23.10 parses prepositional phrases. Noun phrases may contain prepositional phrases and vice versa, so the two networks each contain a push arc which calls the other.

The noun phrase network contains six nodes. The first node, `np` has three choices. If it reads a pronoun, then it can move to node `pron`, which pops out of the network:

```
(defnode np
  (cat det np/det
    (setr det *))
  (jump np/det
    (setr det nil))
  (cat pron  pron
    (setr n *)))

(defnode pron
  (up `(np (pronoun ,(getr n)))))

(defnode np/det
  (down mods np/mods
    (setr mods *))
  (jump np/mods
    (setr mods nil)))

(defnode np/mods
  (cat n np/n
    (setr n *)))

(defnode np/n
  (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))))
  (down pp np/pp
    (setr pp *)))

(defnode np/pp
  (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))
           ,(getr pp))))
```

Figure 23.9: Noun phrase sub-network.

```
> (with-parses np '(it)
    (format t "Parsing: ~A~%" parse))
Parsing: (NP (PRONOUN IT))
@
```

```
(defnode pp
  (cat prep pp/prep
    (setr prep *)))

(defnode pp/prep
  (down np pp/np
    (setr op *)))

(defnode pp/np
  (up `(pp (prep ,(getr prep))
           (obj ,(getr op)))))
```

Figure 23.10: Prepositional phrase sub-network.

Both the other arcs lead to node `np/det`: one arc reads a determiner (e.g. “the”), and the other arc simply jumps, reading no input. At node `np/det`, both arcs lead to `np/mods`; `np/det` has the option of pushing to sub-network `mods` to pick up a string of modifiers, or jumping. Node `np-mods` reads a noun and continues to `np/n`. This node can either pop a result, or push to the prepositional phrase network to try to pick up a prepositional phrase. The final node, `np/pp`, pops a result.

Different types of noun phrases will have different parse paths. Here are two parsings on the noun phrase network:

```
> (with-parses np '(arrows)
    (pprint parse))
(NP (DET NIL)
    (MODIFIERS NIL)
    (NOUN ARROWS))
@
> (with-parses np '(a time fly like him)
    (pprint parse))
(NP (DET A)
    (MODIFIERS (N-GROUP TIME))
    (NOUN FLY)
    (PP (PREP LIKE)
        (OBJ (NP (PRONOUN HIM)))))
@
```

The first parse succeeds by jumping to `np/det`, jumping again to `np/mods`, reading a noun, then popping at `np/n`. The second never jumps, pushing first for

```
(defnode s
  (down np s/subj
    (setr mood 'decl)
    (setr subj *))
  (cat v v
    (setr mood 'imp)
    (setr subj '(np (pron you)))
    (setr aux nil)
    (setr v *)))

(defnode s/subj
  (cat v v
    (setr aux nil)
    (setr v *)))

(defnode v
  (up `(s (mood ,(getr mood))
          (subj ,(getr subj))
          (vcl (aux ,(getr aux))
               (v ,(getr v)))))
  (down np s/obj
    (setr obj *)))

(defnode s/obj
  (up `(s (mood ,(getr mood))
          (subj ,(getr subj))
          (vcl (aux ,(getr aux))
               (v ,(getr v)))
          (obj ,(getr obj)))))
```

Figure 23.11: Sentence network.

a string of modifiers, and again for a prepositional phrase. As is often the case with parsers, expressions which are syntactically well-formed are such nonsense semantically that it's difficult for humans even to detect the syntactic structure. Here the noun phrase "a time fly like him" has the same form as "a Lisp hacker like him."

Now all we need is a network for recognizing sentence structure. The network shown in Figure 23.11 parses both commands and statements. The start node is conventionally called `s`. The first node leaving it pushes for a noun phrase,

```
> (with-parses s '(time flies like an arrow)
    (pprint parse))

(S (MOOD DECL)
   (SUBJ (NP (DET NIL)
             (MODIFIERS (N-GROUP TIME))
             (NOUN FLIES)))
   (VCL (AUX NIL)
        (V LIKE))
   (OBJ (NP (DET AN)
            (MODIFIERS NIL)
            (NOUN ARROW))))

(S (MOOD IMP)
   (SUBJ (NP (PRON YOU)))
   (VCL (AUX NIL)
        (V TIME))
   (OBJ (NP (DET NIL)
            (MODIFIERS NIL)
            (NOUN FLIES)
            (PP (PREP LIKE)
                (OBJ (NP (DET AN)
                         (MODIFIERS NIL)
                         (NOUN ARROW)))))))
@
```

Figure 23.12: Two parsings for a sentence.

which will be the subject of the sentence. The second outgoing arc reads a verb. When a sentence is syntactically ambiguous, both arcs could succeed, ultimately yielding two or more parsings, as in Figure 23.12. The first parsing is analogous to "Island nations like a navy," and the second is analogous to "Find someone like a policeman." More complex ATNs are able to find six or more parsings for "Time flies like an arrow."

The ATN compiler in this chapter is presented more as a distillation of the idea of an ATN than as production software. A few obvious changes would make this code much more efficient. When speed is important, the whole idea of simulating nondeterminism with closures may be too slow. But when it isn't essential, the programming techniques described here lead to very concise programs.

24

Prolog

This chapter describes how to write Prolog as an embedded language. Chapter 19 showed how to write a program which answered complex queries on databases. Here we add one new ingredient: rules, which make it possible to infer facts from those already known. A set of rules defines a tree of implications. In order to use rules which would otherwise imply an unlimited number of facts, we will search this implication tree nondeterministically.

Prolog makes an excellent example of an embedded language. It combines three ingredients: pattern-matching, nondeterminism, and rules. Chapters 18 and 22 give us the first two independently. By building Prolog on top of the pattern-matching and nondeterministic choice operators we have already, we will have an example of a real, multi-layer bottom-up system. Figure 24.1 shows the layers of abstraction involved.

The secondary aim of this chapter is to study Prolog itself. For experienced programmers, the most convenient explanation of Prolog may be a sketch of its implementation. Writing Prolog in Lisp is particularly interesting, because it brings out the similarities between the two languages.

24.1 Concepts

Chapter 19 showed how to write a database system which would accept complex queries containing variables, and generate all the bindings which made the query true in the database. In the following example, (after calling `clear-db`) we assert two facts and then query the database:

Figure 24.1: Layers of abstraction.

```
> (fact painter reynolds)
(REYNOLDS)
> (fact painter gainsborough)
(GAINSBOROUGH)
> (with-answer (painter ?x)
    (print ?x))
GAINSBOROUGH
REYNOLDS
NIL
```

Conceptually, Prolog is the database program with the addition of *rules,* which make it possible to satisfy a query not just by looking it up in the database, but by inferring it from other known facts. For example, if we have a rule like:

If `(hungry ?x)` and `(smells-of ?x turpentine)`
Then `(painter ?x)`

then the query `(painter ?x)` will be satisfied for `?x` = `raoul` when the database contains both `(hungry raoul)` and `(smells-of raoul turpentine)`, even if it doesn't contain `(painter raoul)`.

In Prolog, the if-part of a rule is called the *body,* and the then-part the *head.* (In logic, the names are *antecedent* and *consequent,* but it is just as well to have separate names, to emphasize that Prolog inference is not the same as logical implication.) When trying to establish bindings[1] for a query, the program looks first at the head of a rule. If the head matches the query that the program is trying to answer, the program will then try to establish bindings for the body of the rule. Bindings which satisfy the body will, by definition, satisfy the head.

The facts used in the body of the rule may in turn be inferred from other rules:

[1] Many of the concepts used in this chapter, including this sense of bindings, are explained in Section 18.4.

If (gaunt ?x) or (eats-ravenously ?x)
Then (hungry ?x)

and rules may be recursive, as in:

If (surname ?f ?n) and (father ?f ?c)
Then (surname ?c ?n)

Prolog will be able to establish bindings for a query if it can find some path through the rules which leads eventually to known facts. So it is essentially a search engine: it traverses the tree of logical implications formed by the rules, looking for a successful path.

Though rules and facts sound like distinct types of objects, they are conceptually interchangeable. Rules can be seen as virtual facts. If we want our database to reflect the discovery that big, fierce animals are rare, we could look for all the *x* such that there are facts (species *x*), (big *x*), and (fierce *x*), and add a new fact (rare *x*). However, by defining a rule to say

If (species ?x) and (big ?x) and (fierce ?x)
Then (rare ?x)

we get the same effect, without actually having to add all the (rare *x*) to the database. We can even define rules which imply an infinite number of facts. Thus rules make the database smaller at the expense of extra processing when it comes time to answer questions.

Facts, meanwhile, are a degenerate case of rules. The effect of any fact *F* could be duplicated by a rule whose body was always true:

If true
Then *F*

To simplify our implementation, we will take advantage of this principle and represent facts as bodyless rules. ∘

24.2 An Interpreter

Section 18.4 showed two ways to define if-match. The first was simple but inefficient. Its successor was faster because it did much of its work at compile-time. We will follow a similar strategy here. In order to introduce some of the topics involved, we will begin with a simple interpreter. Later we will show how to write the same program much more efficiently.

```
(defmacro with-inference (query &body body)
 `(progn
    (setq *paths* nil)
    (=bind (binds) (prove-query ',(rep_ query) nil)
      (let ,(mapcar #'(lambda (v)
                        `(,v (fullbind ',v binds)))
                    (vars-in query #'atom))
        ,@body
        (fail)))))

(defun rep_ (x)
  (if (atom x)
      (if (eq x '_) (gensym "?") x)
      (cons (rep_ (car x)) (rep_ (cdr x)))))

(defun fullbind (x b)
  (cond ((varsym? x) (aif2 (binding x b)
                           (fullbind it b)
                           (gensym)))
        ((atom x) x)
        (t (cons (fullbind (car x) b)
                 (fullbind (cdr x) b)))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

Figure 24.2: Toplevel macro.

Figures 24.2–24.4 contain the code for a simple Prolog interpreter. It accepts the same queries as the query interpreter of Section 19.3, but uses rules instead of the database to generate bindings. The query interpreter was invoked through a macro called `with-answer`. The interface to the Prolog interpreter will be through a similar macro, called `with-inference`. Like `with-answer`, `with-inference` is given a query and a series of Lisp expressions. Variables in the query are symbols beginning with a question mark:

```
(with-inference (painter ?x)
  (print ?x))
```

A call to `with-inference` expands into code that will evaluate the Lisp expressions for each set of bindings generated by the query. The call above, for example, will print each *x* for which it is possible to infer (painter *x*).

Figure 24.2 shows the definition of `with-inference`, together with the function it calls to retrieve bindings. One notable difference between `with-answer` and `with-inference` is that the former simply collected all the valid bindings. The new program searches nondeterministically. We see this in the definition of `with-inference`: instead of expanding into a loop, it expands into code which will return one set of bindings, followed by a `fail` to restart the search. This gives us iteration implicitly, as in:

```
> (choose-bind x '(0 1 2 3 4 5 6 7 8 9)
    (princ x)
    (if (= x 6) x (fail)))
0123456
6
```

The function `fullbind` points to another difference between `with-answer` and `with-inference`. Tracing back through a series of rules can build up binding lists in which the binding of a variable is a list of other variables. To make use of the results of a query we now need a recursive function for retrieving bindings. This is the purpose of `fullbind`:

```
> (setq b '((?x . (?y . ?z)) (?y . foo) (?z . nil)))
((?X ?Y . ?Z) (?Y . FOO) (?Z))
> (values (binding '?x b))
(?Y . ?Z)
> (fullbind '?x b)
(FOO)
```

Bindings for the query are generated by a call to `prove-query` in the expansion of `with-inference`. Figure 24.3 shows the definition of this function and the functions it calls. This code is structurally isomorphic to the query interpreter described in Section 19.3. Both programs use the same functions for matching, but where the query interpreter used mapping or iteration, the Prolog interpreter uses equivalent *choose*s.

Using nondeterministic search instead of iteration does make the interpretation of negated queries a bit more complex. Given a query like

```
(not (painter ?x))
```

the query interpreter could just try to establish bindings for `(painter ?x)`, returning `nil` if any were found. With nondeterministic search we have to be more careful: we don't want the interpretation of `(painter ?x)` to fail back outside the scope of the `not`, nor do we want it to leave saved paths that might

```
(=defun prove-query (expr binds)
  (case (car expr)
    (and  (prove-and (cdr expr) binds))
    (or   (prove-or (cdr expr) binds))
    (not  (prove-not (cadr expr) binds))
    (t    (prove-simple expr binds))))

(=defun prove-and (clauses binds)
  (if (null clauses)
      (=values binds)
      (=bind (binds) (prove-query (car clauses) binds)
        (prove-and (cdr clauses) binds))))

(=defun prove-or (clauses binds)
  (choose-bind c clauses
    (prove-query c binds)))

(=defun prove-not (expr binds)
  (let ((save-paths *paths*))
    (setq *paths* nil)
    (choose (=bind (b) (prove-query expr binds)
              (setq *paths* save-paths)
              (fail))
            (progn
              (setq *paths* save-paths)
              (=values binds)))))

(=defun prove-simple (query binds)
  (choose-bind r *rlist*
    (implies r query binds)))
```

Figure 24.3: Interpretation of queries.

be restarted later. So now the test for `(painter ?x)` is done with a temporarily empty list of saved states, and the old list is restored on the way out.

Another difference between this program and the query interpreter is in the interpretation of simple patterns—expressions such as `(painter ?x)` which consist just of a predicate and some arguments. When the query interpreter generated bindings for a simple pattern, it called `lookup` (page 251). Now, instead of calling `lookup`, we have to get any bindings implied by the rules.

```
(defvar *rlist* nil)

(defmacro <- (con &rest ant)
  (let ((ant (if (= (length ant) 1)
                 (car ant)
                 `(and ,@ant))))
    `(length (conc1f *rlist* (rep_ (cons ',ant ',con))))))

(=defun implies (r query binds)
  (let ((r2 (change-vars r)))
    (aif2 (match query (cdr r2) binds)
          (prove-query (car r2) it)
          (fail))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v)
                      (cons v (symb '? (gensym))))
                  (vars-in r #'atom))
          r))
```

Figure 24.4: Code involving rules.

⟨rule⟩	: `(<-` ⟨sentence⟩ ⟨query⟩`)`
⟨query⟩	: `(not` ⟨query⟩`)`
	: `(and` ⟨query⟩`*)`
	: `(or` ⟨query⟩`*)`
	: ⟨sentence⟩
⟨sentence⟩	: `(`⟨symbol⟩ ⟨argument⟩`*)`
⟨argument⟩	: ⟨variable⟩
	: ⟨symbol⟩
	: ⟨number⟩
⟨variable⟩	: `?`⟨symbol⟩

Figure 24.5: Syntax of rules.

Code for defining and using rules is shown in Figure 24.4. The rules are kept in a global list, `*rlist*`. Each rule is represented as a dotted pair of body and head. At the time a rule is defined, all the underscores are replaced with unique variables.

The definition of `<-` follows three conventions often used in programs of this type:

1. New rules are added to the end rather than the front of the list, so that they will be applied in the order that they were defined.

2. Rules are expressed head first, since that's the order in which the program examines them.

3. Multiple expressions in the body are within an implicit `and`.

The outermost call to `length` in the expansion of `<-` is simply to avoid printing a huge list when `<-` is called from the toplevel.

The syntax of rules is given in Figure 24.5. The head of a rule must be a pattern for a fact: a list of a predicate followed by zero or more arguments. The body may be any query that could be handled by the query interpreter of Chapter 19. Here is the rule from earlier in this chapter:

```
(<- (painter ?x) (and (hungry ?x)
                      (smells-of ?x turpentine)))
```

or just

```
(<- (painter ?x) (hungry ?x)
                 (smells-of ?x turpentine))
```

As in the query interpreter, arguments like `turpentine` do not get evaluated, so they don't have to be quoted.

When `prove-simple` is asked to generate bindings for a query, it nondeterministically chooses a rule and sends both rule and query to `implies`. The latter function then tries to match the query with the head of the rule. If the match succeeds, `implies` will call `prove-query` to establish bindings for the body. Thus we recursively search the tree of implications.

The function `change-vars` replaces all the variables in a rule with fresh ones. An `?x` used in one rule is meant to be independent of one used in another. In order to avoid conflicts with existing bindings, `change-vars` is called each time a rule is used.

For the convenience of the user, it is possible to use `_` (underscore) as a wildcard variable in rules. When a rule is defined, the function `rep_` is called to change each underscore into a real variable. Underscores can also be used in the queries given to `with-inference`.

24.3 Rules

This section shows how to write rules for our Prolog. To start with, here are the two rules from Section 24.1:

```
(<- (painter ?x) (hungry ?x)
                 (smells-of ?x turpentine))

(<- (hungry ?x) (or (gaunt ?x) (eats-ravenously ?x)))
```

If we also assert the following facts:

```
(<- (gaunt raoul))
(<- (smells-of raoul turpentine))
(<- (painter rubens))
```

Then we will get the bindings they generate according to the order in which they were defined:

```
> (with-inference (painter ?x)
    (print ?x))
RAOUL
RUBENS
@
```

The `with-inference` macro has exactly the same restrictions on variable binding as `with-answer`. (See Section 19.4.)

We can write rules which imply that facts of a given form are true for all possible bindings. This happens, for example, when some variable occurs in the head of a rule but not in the body. The rule

```
(<- (eats ?x ?f) (glutton ?x))
```

Says that if `?x` is a glutton, then `?x` eats everything. Because `?f` doesn't occur in the body, we can prove any fact of the form `(eats ?x` *y*`)` simply by establishing a binding for `?x`. If we make a query with a literal value as the second argument to `eats`,

```
> (<- (glutton hubert))
7
> (with-inference (eats ?x spinach)
    (print ?x))
HUBERT
@
```

then any literal value will work. When we give a variable as the second argument:

```
> (with-inference (eats ?x ?y)
    (print (list ?x ?y)))
(HUBERT #:G229)
@
```

we get a gensym back. Returning a gensym as the binding of a variable in the query is a way of signifying that *any* value would be true there. Programs can be written explicitly to take advantage of this convention:

```
> (progn
    (<- (eats monster bad-children))
    (<- (eats warhol candy)))
9
> (with-inference (eats ?x ?y)
    (format t "~A eats ~A.~%"
            ?x
            (if (gensym? ?y) 'everything ?y)))
HUBERT eats EVERYTHING.
MONSTER eats BAD-CHILDREN.
WARHOL eats CANDY.
@
```

Finally, if we want to specify that facts of a certain form will be true for *any* arguments, we make the body a conjunction with no arguments. The expression `(and)` will always behave as a true fact. In the macro `<-` (Figure 24.4), the body defaults to `(and)`, so for such rules we can simply omit the body:

```
> (<- (identical ?x ?x))
10
> (with-inference (identical a ?x)
    (print ?x))
A
@
```

For readers with some knowledge of Prolog, Figure 24.6 shows the translation from Prolog syntax into that of our program. The traditional first Prolog program is `append`, which would be written as at the end of Figure 24.6. In an instance of appending, two shorter lists are joined together to form a single larger one. Any two of these lists define what the third should be. The Lisp function `append` takes the two shorter lists as arguments and returns the longer one. Prolog `append` is more general; the two rules in Figure 24.6 define a program which, given any two of the lists involved, can find the third.

Our syntax differs from traditional Prolog syntax as follows:

1. Variables are represented by symbols beginning with question marks instead of capital letters. Common Lisp is not case-sensitive by default, so it would be more trouble than it's worth to use capitals.
2. `[ ]` becomes `nil`.
3. Expressions of the form `[`*x* `|` *y*`]` become `(`*x* `.` *y*`)`.
4. Expressions of the form `[`*x*`,` *y*`, ...]` become `(`*x* *y* `...)`.
5. Predicates are moved inside parentheses, and no commas separate arguments: *pred*`(`*x*`,` *y*`, ...)` becomes `(`*pred* *x* *y* `...)`.

Thus the Prolog definition of `append`:

```
append([ ], Xs, Xs).
append([X | Xs], Ys, [X | Zs]) <- append(Xs, Ys, Zs).
```

becomes:

```
(<- (append nil ?xs ?xs))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
    (append ?xs ?ys ?zs))
```

Figure 24.6: Prolog syntax equivalence.

```
> (with-inference (append ?x (c d) (a b c d))
    (format t "Left: ~A~%" ?x))
Left: (A B)
@
> (with-inference (append (a b) ?x (a b c d))
    (format t "Right: ~A~%" ?x))
Right: (C D)
@
> (with-inference (append (a b) (c d) ?x)
    (format t "Whole: ~A~%" ?x))
Whole: (A B C D)
@
```

Not only that, but given only the last list, it can find all the possibilities for the first two:

```
> (with-inference (append ?x ?y (a b c))
    (format t "Left: ~A  Right: ~A~%" ?x ?y))
Left: NIL  Right: (A B C)
Left: (A)  Right: (B C)
Left: (A B)  Right: (C)
Left: (A B C)  Right: NIL
@
```

The case of append points to a great difference between Prolog and other languages. A collection of Prolog rules does not have to yield a specific value. It can instead yield *constraints,* which, when combined with constraints generated by other parts of the program, yield a specific value. For example, if we define member thus:

```
(<- (member ?x (?x . ?rest)))
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
```

then we can use it to test for list membership, as we would use the Lisp function member:

```
> (with-inference (member a (a b)) (print t))
T
@
```

but we can also use it to establish a constraint of membership, which, combined with other constraints, yields a specific list. If we also have a predicate cara

```
(<- (cara (a _)))
```

which is true of any two-element list whose car is a, then between that and member we have enough constraint for Prolog to construct a definite answer:

```
> (with-inference (and (cara ?lst) (member b ?lst))
    (print ?lst))
(A B)
@
```

This is a rather trivial example, but bigger programs can be constructed on the same principle. Whenever we want to program by combining partial solutions, Prolog may be useful. Indeed, a surprising variety of problems can be expressed in such terms: Figure 24.14, for example, shows a sorting algorithm expressed as a collection of constraints on the solution.

24.4 The Need for Nondeterminism

Chapter 22 explained the relation between deterministic and nondeterministic search. A deterministic search program could take a query and generate all the solutions which satisfied it. A nondeterministic search program will use *choose* to generate solutions one at a time, and if more are needed, will call *fail* to restart the search.

When we have rules which all yield finite sets of bindings, and we want all of them at once, there is no reason to prefer nondeterministic search. The difference between the two strategies becomes apparent when we have queries which would generate an infinite number of bindings, of which we want a finite subset. For example, the rules

```
(<- (all-elements ?x nil))
(<- (all-elements ?x (?x . ?rest))
    (all-elements ?x ?rest))
```

imply all the facts of the form `(all-elements` *x y*`)`, where every member of *y* is equal to *x*. Without backtracking we could handle queries like:

```
(all-elements a  (a a a))
(all-elements a  (a a b))
(all-elements ?x (a a a))
```

However, the query `(all-elements a ?x)` is satisfied for an infinite number of possible `?x`: `nil`, `(a)`, `(a a)`, and so on. If we try to generate answers for this query by iteration, the iteration will never terminate. Even if we only wanted one of the answers, we would never get a result from an implementation which had to generate all the bindings for the query before it could begin to iterate through the Lisp expressions following it.

This is why `with-inference` interleaves the generation of bindings with the evaluation of its body. Where queries could lead to an infinite number of answers, the only successful approach will be to generate answers one at a time, and return to pick up new ones by restarting the suspended search. Because it uses *choose* and *fail*, our program can handle this case:

```
> (block nil
    (with-inference (all-elements a ?x)
      (if (= (length ?x) 3)
          (return ?x)
          (princ ?x))))
NIL(A)(A A)
(A A A)
```

Like any other Prolog implementation, ours simulates nondeterminism by doing depth-first search with backtracking. In theory, "logic programs" run under true nondeterminism. In fact, Prolog implementations always use depth-first search. Far from being inconvenienced by this choice, typical Prolog programs depend on it. In a truly nondeterministic world, the query

```
(and (all-elements a ?x) (length ?x 3))
```

has an answer, but it takes you arbitrarily long to find out what it is.

Not only does Prolog use the depth-first implementation of nondeterminism, it uses a version equivalent to that defined on page 293. As explained there, this implementation is not always guaranteed to terminate. So Prolog programmers must take deliberate steps to avoid loops in the search space. For example, if we had defined `member` in the reverse order

```
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
(<- (member ?x (?x . ?rest)))
```

then logically it would have the same meaning, but as a Prolog program it would have a different effect. The original definition of `member` would yield an infinite stream of answers in response to the query `(member 'a ?x)`, but the reversed definition will yield an infinite recursion, and no answers.

24.5 New Implementation

In this section we will see another instance of a familiar pattern. In Section 18.4, we found after writing the initial version that `if-match` could be made much faster. By taking advantage of information known at compile-time, we were able to write a new version which did less work at runtime. We saw the same phenomenon on a larger scale in Chapter 19. Our query interpreter was replaced by an equivalent but faster version. The same thing is about to happen to our Prolog interpreter.

Figures 24.7, 24.8, and 24.10 define Prolog in a different way. The macro `with-inference` used to be just the interface to a Prolog interpreter. Now it is most of the program. The new program has the same general shape as the old one, but of the functions defined in Figure 24.8, only `prove` is called at runtime. The others are called by `with-inference` in order to generate its expansion.

Figure 24.7 shows the new definition of `with-inference`. As in `if-match` or `with-answer`, pattern variables are initially bound to gensyms to indicate that they haven't yet been assigned real values by matching. Thus the function `varsym?`, which `match` and `fullbind` use to detect variables, has to be changed to look for gensyms.

```
(defmacro with-inference (query &rest body)
  (let ((vars (vars-in query #'simple?)) (gb (gensym)))
    `(with-gensyms ,vars
       (setq *paths* nil)
       (=bind (,gb) ,(gen-query (rep_ query))
         (let ,(mapcar #'(lambda (v)
                            `(,v (fullbind ,v ,gb)))
                       vars)
           ,@body)
         (fail)))))

(defun varsym? (x)
  (and (symbolp x) (not (symbol-package x))))
```

Figure 24.7: New toplevel macro.

To generate the code to establish bindings for the query, `with-inference` calls `gen-query` (Figure 24.8). The first thing `gen-query` does is look to see whether its first argument is a complex query beginning with an operator like `and` or `or`. This process continues recursively until it reaches simple queries, which are expanded into calls to `prove`. In the original implementation, such logical structure was analyzed at runtime. A complex expression occurring in the body of a rule had to be analyzed anew each time the rule was used. This is wasteful because the logical structure of rules and queries is known beforehand. The new implementation decomposes complex expressions at compile-time.

As in the previous implementation, a `with-inference` expression expands into code which iterates through the Lisp code following the query with the pattern variables bound to successive values established by the rules. The expansion of `with-inference` concludes with a `fail`, which will restart any saved states.

The remaining functions in Figure 24.8 generate expansions for complex queries—queries joined together by operators like `and`, `or`, and `not`. If we have a query like

```
(and (big ?x) (red ?x))
```

then we want the Lisp code to be evaluated only with those `?x` for which both conjuncts can be proved. So to generate the expansion of an `and`, we nest the expansion of the second conjunct within that of the first. When `(big ?x)` succeeds we try `(red ?x)`, and if that succeeds, we evaluate the Lisp expressions. So the whole expression expands as in Figure 24.9.

```
(defun gen-query (expr &optional binds)
  (case (car expr)
    (and (gen-and (cdr expr) binds))
    (or  (gen-or  (cdr expr) binds))
    (not (gen-not (cadr expr) binds))
    (t   `(prove (list ',(car expr)
                       ,@(mapcar #'form (cdr expr)))
                 ,binds))))

(defun gen-and (clauses binds)
  (if (null clauses)
      `(=values ,binds)
      (let ((gb (gensym)))
        `(=bind (,gb) ,(gen-query (car clauses) binds)
           ,(gen-and (cdr clauses) gb)))))

(defun gen-or (clauses binds)
  `(choose
     ,@(mapcar #'(lambda (c) (gen-query c binds))
               clauses)))

(defun gen-not (expr binds)
  (let ((gpaths (gensym)))
    `(let ((,gpaths *paths*))
       (setq *paths* nil)
       (choose (=bind (b) ,(gen-query expr binds)
                 (setq *paths* ,gpaths)
                 (fail))
               (progn
                 (setq *paths* ,gpaths)
                 (=values ,binds))))))

(=defun prove (query binds)
   (choose-bind r *rules* (=funcall r query binds)))

(defun form (pat)
  (if (simple? pat)
      pat
      `(cons ,(form (car pat)) ,(form (cdr pat)))))
```

Figure 24.8: Compilation of queries.

```
(with-inference (and (big ?x) (red ?x))
  (print ?x))
```

expands into:

```
(with-gensyms (?x)
  (setq *paths* nil)
  (=bind (#:g1) (=bind (#:g2) (prove (list 'big ?x) nil)
                  (=bind (#:g3) (prove (list 'red ?x) #:g2)
                    (=values #:g3)))
     (let ((?x (fullbind ?x #:g1)))
       (print ?x))
     (fail)))
```

Figure 24.9: Expansion of a conjunction.

An `and` means nesting; an `or` means a *choose*. Given a query like

```
(or (big ?x) (red ?x))
```

we want the Lisp expressions to be evaluated for values of `?x` established by either subquery. The function `gen-or` expands into a `choose` over the `gen-query` of each of the arguments. As for `not`, `gen-not` is almost identical to `prove-not` (Figure 24.3).

Figure 24.10 shows the code for defining rules. Rules are translated directly into Lisp code generated by `rule-fn`. Since `<-` now expands rules into Lisp code, compiling a file full of rule definitions will cause rules to be compiled functions.

When a rule-function is sent a pattern, it tries to match it with the head of the rule it represents. If the match succeeds, the rule-function will then try to establish bindings for the body. This task is essentially the same as that done by `with-inference`, and in fact `rule-fn` ends by calling `gen-query`. The rule-function eventually returns the bindings established for the variables occurring in the head of the rule.

24.6 Adding Prolog Features

The code already presented can run most "pure" Prolog programs. The final step is to add extras like cuts, arithmetic, and I/O.

Putting a *cut* in a Prolog rule causes the search tree to be pruned. Ordinarily, when our program encounters a `fail`, it backtracks to the last choice point. The

```
(defvar *rules* nil)

(defmacro <- (con &rest ant)
  (let ((ant (if (= (length ant) 1)
                 (car ant)
                 `(and ,@ant))))
    `(length (conc1f *rules*
                     ,(rule-fn (rep_ ant) (rep_ con))))))

(defun rule-fn (ant con)
  (with-gensyms (val win fact binds)
    `(=lambda (,fact ,binds)
       (with-gensyms ,(vars-in (list ant con) #'simple?)
         (multiple-value-bind
             (,val ,win)
             (match ,fact
                    (list ',(car con)
                          ,@(mapcar #'form (cdr con)))
                    ,binds)
           (if ,win
               ,(gen-query ant val)
               (fail)))))))
```

Figure 24.10: Code for defining rules.

implementation of *choose* in Section 22.4 stores choice points in the global variable `*paths*`. Calling `fail` restarts the search at the most recent choice point, which is the car of `*paths*`. Cuts introduce a new complication. When the program encounters a `cut`, it will throw away some of the most recent choice points stored on `*paths*`—specifically, all those stored since the last call to `prove`.

The effect is to make rules mutually exclusive. We can use cuts to get the effect of a case statement in Prolog programs. For example, if we define `minimum` this way:

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)))
(<- (minimum ?x ?y ?y) (lisp (> ?x ?y)))
```

it will work correctly, but inefficiently. Given the query

```
(minimum 1 2 ?x)
```

Prolog will immediately establish that `?x` = 1 from the first rule. A human would stop here, but the program will waste time looking for more answers from the second rule, because it has been given no indication that the two rules are mutually exclusive. On the average, this version of `minimum` will do 50% more work than it needs to. We can fix the problem by adding a cut after the first test:

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)) (cut))
(<- (minimum ?x ?y ?y))
```

Now when Prolog has finished with the first rule, it will fail all the way out of the query instead of moving on to the next rule.

It is trivially easy to modify our program to handle cuts. On each call to `prove`, the current state of `*paths*` is passed as a parameter. If the program encounters a cut, it just sets `*paths*` back to the old value passed in the parameter. Figures 24.11 and 24.12 show the code which has to be modified to handle cuts. (Changed lines are marked with semicolons. Not all the changes are due to cuts.)

Cuts which merely make a program more efficient are called *green cuts.* The cut in `minimum` was a green cut. Cuts which make a program behave differently are called *red cuts.* For example, if we define the predicate `artist` as follows:

```
(<- (artist ?x) (sculptor ?x) (cut))
(<- (artist ?x) (painter ?x))
```

the result is that, if there are any sculptors, then the query can end there. If there are no sculptors then painters get to be considered as artists:

```
> (progn (<- (painter 'klee))
         (<- (painter 'soutine)))
4
> (with-inference (artist ?x)
    (print ?x))
KLEE
SOUTINE
@
```

But if there are sculptors, the cut stops inference with the first rule:

```
> (<- (sculptor 'hepworth))
5
> (with-inference (artist ?x)
    (print ?x))
HEPWORTH
@
```

```
(defun rule-fn (ant con)
  (with-gensyms (val win fact binds paths)                              ;
    `(=lambda (,fact ,binds ,paths)                                     ;
       (with-gensyms ,(vars-in (list ant con) #'simple?)
         (multiple-value-bind
             (,val ,win)
             (match ,fact
                    (list ',(car con)
                          ,@(mapcar #'form (cdr con)))
                    ,binds)
           (if ,win
               ,(gen-query ant val paths)                               ;
               (fail)))))))

(defmacro with-inference (query &rest body)
  (let ((vars (vars-in query #'simple?)) (gb (gensym)))
    `(with-gensyms ,vars
       (setq *paths* nil)
       (=bind (,gb) ,(gen-query (rep_ query) nil '*paths*) ;
         (let ,(mapcar #'(lambda (v)
                           `(,v (fullbind ,v ,gb)))
                       vars)
           ,@body)
         (fail)))))

(defun gen-query (expr binds paths)                                    ;
  (case (car expr)
    (and  (gen-and (cdr expr) binds paths))                            ;
    (or   (gen-or  (cdr expr) binds paths))                            ;
    (not  (gen-not (cadr expr) binds paths))                           ;
    (lisp (gen-lisp (cadr expr) binds))                                ;
    (is   (gen-is (cadr expr) (third expr) binds))                     ;
    (cut  `(progn (setq *paths* ,paths)                                ;
                  (=values ,binds)))                                   ;
    (t    `(prove (list ',(car expr)
                        ,@(mapcar #'form (cdr expr)))
                  ,binds *paths*))))                                   ;

(=defun prove (query binds paths)                                      ;
   (choose-bind r *rules*
     (=funcall r query binds paths)))                                  ;
```

Figure 24.11: Adding support for new operators.

```
(defun gen-and (clauses binds paths)                         ;
  (if (null clauses)
      `(=values ,binds)
      (let ((gb (gensym)))
       `(=bind (,gb) ,(gen-query (car clauses) binds paths);
          ,(gen-and (cdr clauses) gb paths)))))              ;

(defun gen-or (clauses binds paths)                          ;
  `(choose
     ,@(mapcar #'(lambda (c) (gen-query c binds paths))     ;
               clauses)))

(defun gen-not (expr binds paths)                            ;
  (let ((gpaths (gensym)))
    `(let ((,gpaths *paths*))
       (setq *paths* nil)
       (choose (=bind (b) ,(gen-query expr binds paths)     ;
                 (setq *paths* ,gpaths)
                 (fail))
               (progn
                 (setq *paths* ,gpaths)
                 (=values ,binds))))))

(defmacro with-binds (binds expr)
  `(let ,(mapcar #'(lambda (v) `(,v (fullbind ,v ,binds)))
                 (vars-in expr))
     ,expr))

(defun gen-lisp (expr binds)
  `(if (with-binds ,binds ,expr)
       (=values ,binds)
       (fail)))

(defun gen-is (expr1 expr2 binds)
  `(aif2 (match ,expr1 (with-binds ,binds ,expr2) ,binds)
         (=values it)
         (fail)))
```

Figure 24.12: Adding support for new operators.

⟨rule⟩	: (<- ⟨sentence⟩ ⟨query⟩)
⟨query⟩	: (not ⟨query⟩)
	: (and ⟨query⟩*)
	: (lisp ⟨lisp expression⟩)
	: (is ⟨variable⟩ ⟨lisp expression⟩)
	: (cut)
	: (fail)
	: ⟨sentence⟩
⟨sentence⟩	: (⟨symbol⟩ ⟨argument⟩*)
⟨argument⟩	: ⟨variable⟩
	: ⟨lisp expression⟩
⟨variable⟩	: ?⟨symbol⟩

Figure 24.13: New syntax of rules.

The cut is sometimes used in conjunction with the Prolog fail operator. Our function `fail` does exactly the same thing. Putting a cut in a rule makes it like a one-way street: once you enter, you're committed to using only that rule. Putting a cut-fail combination in a rule makes it like a one-way street in a dangerous neighborhood: once you enter, you're committed to leaving with nothing. A typical example is in the implementation of not-equal:

```
(<- (not-equal ?x ?x) (cut) (fail))
(<- (not-equal ?x ?y))
```

The first rule here is a trap for impostors. If we're trying to prove a fact of the form `(not-equal 1 1)`, it will match with the head of the first rule and thus be doomed. The query `(not-equal 1 2)`, on the other hand, will not match the head of the first rule, and will go on to the second, where it succeeds:

```
> (with-inference (not-equal 'a 'a)
    (print t))
@
> (with-inference (not-equal '(a a) '(a b))
    (print t))
T
@
```

The code shown in Figures 24.11 and 24.12 also gives our program arithmetic, I/O, and the Prolog `is` operator. Figure 24.13 shows the complete syntax of rules and queries.

We add arithmetic (and more) by including a trapdoor to Lisp. Now in addition to operators like `and` and `or`, we have the `lisp` operator. This may be followed by any Lisp expression, which will be evaluated with the variables within it bound to the bindings established for them by the query. If the expression evaluates to `nil`, then the `lisp` expression as a whole is equivalent to a `(fail)`; otherwise it is equivalent to `(and)`.

As an example of the use of the `lisp` operator, consider the Prolog definition of `ordered`, which is true of lists whose elements are arranged in ascending order:

```
(<- (ordered (?x)))
(<- (ordered (?x ?y . ?ys))
    (lisp (<= ?x ?y))
    (ordered (?y . ?ys)))
```

In English, a list of one element is ordered, and a list of two or more elements is ordered if the first element of the list is less than or equal to the second, and the list from the second element on is ordered.

```
> (with-inference (ordered '(1 2 3))
    (print t))
T
@
> (with-inference (ordered '(1 3 2))
    (print t))
@
```

By means of the `lisp` operator we can provide other features offered by typical Prolog implementations. Prolog I/O predicates can be duplicated by putting Lisp I/O calls within `lisp` expressions. The Prolog `assert`, which as a side-effect defines new rules, can be duplicated by calling the `<-` macro within `lisp` expressions.

The `is` operator offers a form of assignment. It takes two arguments, a pattern and a Lisp expression, and tries to match the pattern with the result returned by the expression. If the match fails, then the program calls `fail`; otherwise it proceeds with the new bindings. Thus, the expression `(is ?x 1)` has the effect of setting `?x` to `1`, or more precisely, insisting that `?x` be `1`. We need `is` to *calculate*—for example, to calculate factorials:

```
(<- (factorial 0 1))
(<- (factorial ?n ?f)
    (lisp (> ?n 0))
    (is ?n1 (- ?n 1))
    (factorial ?n1 ?f1)
    (is ?f (* ?n ?f1)))
```

We use this definition by making a query with a number *n* as the first argument and a variable as the second:

```
> (with-inference (factorial 8 ?x)
    (print ?x))
40320
@
```

Note that the variables used in a `lisp` expression, or in the second argument to `is`, must have established bindings for the expression to return a value. This restriction holds in any Prolog. For example, the query:

```
(with-inference (factorial ?x 120)                    ; wrong
  (print ?x))
```

won't work with this definition of `factorial`, because `?n` will be unknown when the `lisp` expression is evaluated. So not all Prolog programs are like `append`: many insist, like `factorial`, that certain of their arguments be real values.

24.7 Examples

This final section shows how to write some example Prolog programs in our implementation. The rules in Figure 24.14 define quicksort. These rules imply facts of the form `(quicksort` *x* *y*`)`, where *x* is a list and *y* is a list of the same elements sorted in ascending order. Variables may appear in the second argument position:

```
> (with-inference (quicksort '(3 2 1) ?x)
    (print ?x))
(1 2 3)
@
```

An I/O loop is a test for our Prolog, because it makes use of the `cut`, `lisp`, and `is` operators. The code is shown in Figure 24.15. These rules should be invoked by trying to prove `(echo)`, with no arguments. That query will match the first rule, which will bind `?x` to the result returned by `read`, and then try to establish `(echo ?x)`. The new query can match either of the second two rules. If `?x` = `done`, then the query will terminate in the second rule. Otherwise the query will only match the third rule, which prints the value read, and starts the process over again.

```
(setq *rules* nil)

(<- (append nil ?ys ?ys))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
    (append ?xs ?ys ?zs))

(<- (quicksort (?x . ?xs) ?ys)
    (partition ?xs ?x ?littles ?bigs)
    (quicksort ?littles ?ls)
    (quicksort ?bigs ?bs)
    (append ?ls (?x . ?bs) ?ys))
(<- (quicksort nil nil))

(<- (partition (?x . ?xs) ?y (?x . ?ls) ?bs)
    (lisp (<= ?x ?y))
    (partition ?xs ?y ?ls ?bs))
(<- (partition (?x . ?xs) ?y ?ls (?x . ?bs))
    (lisp (> ?x ?y))
    (partition ?xs ?y ?ls ?bs))
(<- (partition nil ?y nil nil))
```

Figure 24.14: Quicksort.

```
(<- (echo)
    (is ?x (read))
    (echo ?x))
(<- (echo 'done)
    (cut))
(<- (echo ?x)
    (lisp (prog1 t (format t "~A~%" ?x)))
    (is ?y (read))
    (cut)
    (echo ?y))
```

Figure 24.15: An I/O loop in Prolog.

Collectively, the rules define a program that will continue to echo what you type, until you type done:

```
> (with-inference (echo))
hi
HI
ho
HO
done
@
```

Programs like this are difficult to read because they subvert the abstract model of Prolog. It might be easier to understand echo if we look at a literal Lisp translation

```
(defun echo (&rest args)
  (cond ((null args) (echo (read)))
        ((eq (car args) 'done) nil)
        (t (format t "~A~%" (car args))
           (echo (read)))))
```

which in idiomatic Common Lisp would be:

```
(defun echo (&optional (arg (read)))
  (unless (eq arg 'done)
    (format t "~A~%" arg)
    (echo)))
```

24.8 The Senses of Compile

The word "compile" has several senses. In the most general sense, to compile is to transform some abstract description of a program into lower-level code. The program described in this chapter is certainly a compiler in this sense, because it translates rules into Lisp functions.

In a more specific sense, to compile is to transform a program into machine language. Good Common Lisps compile functions into native machine code. As mentioned on page 25, if code which generates closures is compiled, it will yield compiled closures. Thus the program described here is a compiler in the stricter sense as well. In a good Lisp, our Prolog programs will get translated into machine language.

However, the program described here *is still not a Prolog compiler.* For programming languages there is a still more specific sense of "compile," and merely generating machine code is not enough to satisfy this definition. A compiler for a programming language must optimize as well as translate. For example, if a Lisp compiler is asked to compile an expression like

```
(+ x (+ 2 5))
```

it should be smart enough to realize that there is no reason to wait until runtime to evaluate `(+ 2 5)`. The program can be optimized by replacing it with `7`, and instead compiling

```
(+ x 7)
```

In our program, all the compiling is done by the Lisp compiler, and it is looking for Lisp optimizations, not Prolog optimizations. Its optimizations will be valid ones, but too low-level. The Lisp compiler doesn't know that the code it's compiling is meant to represent rules. While a real Prolog compiler would be looking for rules that could be transformed into loops, our program is looking for expressions that yield constants, or closures that could be allocated on the stack.

Embedded languages allow you to make the most of available abstractions, but they are not magic. If you want to travel all the way from a very abstract representation to fast machine code, someone still has to tell the computer how to do it. In this chapter we travelled a good part of that distance with surprisingly little code, but that is not the same as writing a true Prolog compiler.

25

Object-Oriented Lisp

This chapter discusses object-oriented programming in Lisp. Common Lisp includes a set of operators for writing object-oriented programs. Collectively they are called the Common Lisp Object System, or CLOS. Here we consider CLOS not just as a way of writing object-oriented programs, but as a Lisp program itself. Seeing CLOS in this light is the key to understanding the relation between Lisp and object-oriented programming.

25.1 Plus ça Change

Object-oriented programming means a change in the way programs are organized. This change is analogous to the one that has taken place in the distribution of processor power. In 1970, a multi-user computer system meant one or two big mainframes connected to a large number of dumb terminals. Now it is more likely to mean a large number of workstations connected to one another by a network. The processing power of the system is now distributed among individual users instead of centralized in one big computer.

Object-oriented programming breaks up traditional programs in much the same way: instead of having a single program which operates on an inert mass of data, the data itself is told how to behave, and the program is implicit in the interactions of these new data "objects."

For example, suppose we want to write a program to find the areas of two-dimensional shapes. One way to do this would be to write a single function which looked at the type of its argument and behaved accordingly:

```
(defun area (x)
  (cond ((rectangle-p x) (* (height x) (width x)))
        ((circle-p x) (* pi (expt (radius x) 2)))))
```

The object-oriented approach is to make each object able to calculate its own area. The `area` function is broken apart and each clause distributed to the appropriate class of object; the `area` method of the `rectangle` class might be

```
#'(lambda (x) (* (height x) (width x)))
```

and for the `circle` class,

```
#'(lambda (x) (* pi (expt (radius x) 2)))
```

In this model, we ask an object what its area is, and it responds according to the method provided for its class.

The arrival of CLOS might seem a sign that Lisp is changing to embrace the object-oriented paradigm. Actually, it would be more accurate to say that Lisp is staying the same to embrace the object-oriented paradigm. But the principles underlying Lisp don't have a name, and object-oriented programming does, so ○ there is a tendency now to describe Lisp as an object-oriented language. It would be closer to the truth to say that Lisp is an extensible language in which constructs for object-oriented programming can easily be written.

Since CLOS comes pre-written, it is not false advertising to describe Lisp as an object-oriented language. However, it would be limiting to see Lisp as merely that. Lisp is an object-oriented language, yes, but not because it has adopted the object-oriented model. Rather, that model turns out to be just one more permutation of the abstractions underlying Lisp. And to prove it we have CLOS, a program written in Lisp, which makes Lisp an object-oriented language.

The aim of this chapter is to bring out the connection between Lisp and object-oriented programming by studying CLOS as an example of an embedded language. This is also a good way to understand CLOS itself: in the end, nothing explains a language feature more effectively than a sketch of its implementation. In Section 7.6, macros were explained this way. The next section gives a similar sketch of how to build object-oriented abstractions on top of Lisp. This program provides a reference point from which to describe CLOS in Sections 25.3–25.6.

25.2 Objects in Plain Lisp

We can mold Lisp into many different kinds of languages. There is a particularly direct mapping between the concepts of object-oriented programming and the fundamental abstractions of Lisp. The size of CLOS tends to obscure this fact. So before looking at what we can do with CLOS, let's see what we can do with plain Lisp.

Much of what we want from object-oriented programming, we have already in Lisp. We can get the rest with surprisingly little code. In this section, we will define an object system sufficient for many real applications in two pages of code. Object-oriented programming, at a minimum, implies

1. objects which have properties
2. and respond to messages,
3. and which inherit properties and methods from their parents.

In Lisp, there are already several ways to store collections of properties. One way would be to represent objects as hash-tables, and store their properties as entries within them. We then have access to individual properties through `gethash`:

```
(gethash 'color obj)
```

Since functions are data objects, we can store them as properties too. This means that we can also have methods; to invoke a given method of an object is to funcall the property of that name:

```
(funcall (gethash 'move obj) obj 10)
```

We can define a Smalltalk style message-passing syntax upon this idea:

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

so that to tell obj to move 10 we can say

```
(tell obj 'move 10)
```

In fact, the only ingredient plain Lisp lacks is inheritance, and we can provide a rudimentary version of that in six lines of code, by defining a recursive version of `gethash`:

```
(defun rget (obj prop)
  (multiple-value-bind (val win) (gethash prop obj)
    (if win
        (values val win)
        (let ((par (gethash 'parent obj)))
          (and par (rget par prop))))))
```

```
(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (get-ancestors obj)))

(defun get-ancestors (obj)
  (labels ((getall (x)
             (append (list x)
                     (mapcan #'getall
                             (gethash 'parents x)))))
    (stable-sort (delete-duplicates (getall obj))
                 #'(lambda (x y)
                     (member y (gethash 'parents x))))))

(defun some2 (fn lst)
  (if (atom lst)
      nil
      (multiple-value-bind (val win) (funcall fn (car lst))
        (if (or val win)
            (values val win)
            (some2 fn (cdr lst))))))
```

Figure 25.1: Multiple inheritance.

If we just use `rget` in place of `gethash`, we will get inherited properties and methods. We specify an object's parent thus:

```
(setf (gethash 'parent obj) obj2)
```

So far we have only single inheritance—an object can only have one parent. But we can have multiple inheritance by making the `parent` property a list, and defining `rget` as in Figure 25.1.

With single inheritance, when we wanted to retrieve some property of an object, we just searched recursively up its ancestors. If the object itself had no information about the property we wanted, we looked at its parent, and so on. With multiple inheritance we want to perform the same kind of search, but our job is complicated by the fact that an object's ancestors can form a graph instead of a simple list. We can't just search this graph depth-first. With multiple parents we can have the hierarchy shown in Figure 25.2: `a` is descended from `b` and `c`, which are both descended from `d`. A depth-first (or rather, height-first) traversal would go `a`, `b`, `d`, `c`, `d`. If the desired property were present in both `d` and `c`, we would

Figure 25.2: Multiple paths to a superclass.

get the value stored in `d`, not the one stored in `c`. This would violate the principle that subclasses override the default values provided by their parents.

If we want to implement the usual idea of inheritance, we should never examine an object before one of its descendants. In this case, the proper search order would be `a`, `b`, `c`, `d`. How can we ensure that the search always tries descendants first? The simplest way is to assemble a list of all the ancestors of the original object, sort the list so that no object appears before one of its descendants, and then look at each element in turn.

This strategy is used by `get-ancestors`, which returns a properly ordered list of an object and its ancestors. To sort the list, `get-ancestors` calls `stable-sort` instead of `sort`, to avoid the possibility of reordering parallel ancestors. Once the list is sorted, `rget` merely searches for the first object with the desired property. (The utility `some2` is a version of `some` for use with functions like `gethash` that indicate success or failure in the second return value.)

The list of an object's ancestors goes from most specific to least specific: if `orange` is a child of `citrus`, which is a child of `fruit`, then the list will go `(orange citrus fruit)`.

When an object has multiple parents, their precedence goes left-to-right. That is, if we say

```
(setf (gethash 'parents x) (list y z))
```

then `y` will be considered before `z` when we look for an inherited property. For example, we can say that a patriotic scoundrel is a scoundrel first and a patriot second:

```
> (setq scoundrel (make-hash-table)
        patriot (make-hash-table)
        patriotic-scoundrel (make-hash-table))
#<Hash-Table C4219E>
```

```
(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (gethash 'parents obj) parents)
    (ancestors obj)
    obj))

(defun ancestors (obj)
  (or (gethash 'ancestors obj)
      (setf (gethash 'ancestors obj) (get-ancestors obj))))

(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (ancestors obj)))
```

Figure 25.3: A function to create objects.

```
> (setf (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash 'parents patriotic-scoundrel)
                  (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget patriotic-scoundrel 'serves)
SELF
T
```

Let's make some improvements to this skeletal system. We could begin with a function to create objects. This function should build a list of an object's ancestors at the time the object is created. The current code builds these lists when queries are made, but there is no reason not to do it earlier. Figure 25.3 defines a function called `obj` which creates a new object, storing within it a list of its ancestors. To take advantage of stored ancestors, we also redefine `rget`.

Another place for improvement is the syntax of message calls. The `tell` itself is unnecessary clutter, and because it makes verbs come second, it means that our programs can no longer be read like normal Lisp prefix expressions:

```
(tell (tell obj 'find-owner) 'find-owner)
```

We can get rid of the `tell` syntax by defining each property name as a function, as in Figure 25.4. The optional argument `meth?`, if true, signals that this property should be treated as a method. Otherwise it will be treated as a slot, and the value retrieved by `rget` will simply be returned. Once we have defined the name of either kind of property,

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget obj ',name)))
     (defsetf ,name (obj) (val)
       `(setf (gethash ',',name ,obj) ,val))))

(defun run-methods (obj name args)
  (let ((meth (rget obj name)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

Figure 25.4: Functional syntax.

```
(defprop find-owner t)
```

we can refer to it with a function call, and our code will read like Lisp again:

```
(find-owner (find-owner obj))
```

Our previous example now becomes somewhat more readable:

```
> (progn
    (setq scoundrel (obj))
    (setq patriot (obj))
    (setq patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self)
    (setf (serves patriot) 'country)
    (serves patriotic-scoundrel))
SELF
T
```

In the current implementation, an object can have at most one method of a given name. An object either has its own method, or inherits one. It would be convenient to have more flexibility on this point, so that we could combine local and inherited methods. For example, we might want the `move` method of some object to be the `move` method of its parent, but with some extra code run before or afterwards.

To allow for such possibilities, we will modify our program to include before-, after-, and around-methods. Before-methods allow us to say "But first, do this." They are called, most specific first, as a prelude to the rest of the method call. After-methods allow us to say "P.S. Do this too." They are called, most specific last, as an epilogue to the method call. Between them, we run what used to be the whole method, and is now called the *primary method.* The value of this call is the one returned, even if after-methods are called later.

Before- and after-methods allow us to wrap new behavior around the call to the primary method. Around-methods provide a more drastic way of doing the same thing. If an around-method exists, it will be called *instead* of the primary method. Then, at its own discretion, the around-method may itself invoke the primary method (via `call-next`, which will be provided in Figure 25.7).

To allow auxiliary methods, we modify `run-methods` and `rget` as in Figures 25.5 and 25.6. In the previous version, when we ran some method of an object, we ran just one function: the most specific primary method. We ran the first method we encountered when searching the list of ancestors. With auxiliary methods, the calling sequence now goes as follows:

1. The most specific around-method, if there is one.
2. Otherwise, in order:
   (a) All before-methods, from most specific to least specific.
   (b) The most specific primary method (what we used to call).
   (c) All after-methods, from least specific to most specific.

Notice also that instead of being a single function, a method becomes a four-part structure. To define a (primary) method, instead of saying:

```
(setf (gethash 'move obj) #'(lambda ...))
```

we say:

```
(setf (meth-primary (gethash 'move obj)) #'(lambda ...))
```

For this and other reasons, our next step should be to define a macro for defining methods.

Figure 25.7 shows the definition of such a macro. The bulk of this code is taken up with implementing two functions that methods can use to refer to other methods. Around- and primary methods can use `call-next` to invoke the *next method,* which is the code that would have run if the current method didn't exist. For example, if the currently running method is the only around-method, the next

```
(defstruct meth  around before primary after)

(defmacro meth- (field obj)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (and (meth-p ,gobj)
            (,(symb 'meth- field) ,gobj)))))

(defun run-methods (obj name args)
  (let ((pri (rget obj name :primary)))
    (if pri
        (let ((ar (rget obj name :around)))
          (if ar
              (apply ar obj args)
              (run-core-methods obj name args pri)))
        (error "No primary ~A method for ~A." name obj))))

(defun run-core-methods (obj name args &optional pri)
  (multiple-value-prog1
    (progn (run-befores obj name args)
           (apply (or pri (rget obj name :primary))
                  obj args))
    (run-afters obj name args)))

(defun rget (obj prop &optional meth (skip 0))
  (some2 #'(lambda (a)
             (multiple-value-bind (val win) (gethash prop a)
               (if win
                   (case meth (:around  (meth- around val))
                              (:primary (meth- primary val))
                              (t (values val win))))))
         (nthcdr skip (ancestors obj))))
```

Figure 25.5: Auxiliary methods.

method would be the usual sandwich of before-, most specific primary, and after-methods. Within the most specific primary method, the next method would be the second most specific primary method. Since the behavior of `call-next` depends on where it is called, it is never defined globally with a `defun`, but is defined locally within each method defined by `defmeth`.

```
(defun run-befores (obj prop args)
  (dolist (a (ancestors obj))
    (let ((bm (meth- before (gethash prop a))))
      (if bm (apply bm obj args)))))

(defun run-afters (obj prop args)
  (labels ((rec (lst)
             (when lst
               (rec (cdr lst))
               (let ((am (meth- after
                                 (gethash prop (car lst)))))
                 (if am (apply am (car lst) args))))))
    (rec (ancestors obj))))
```

Figure 25.6: Auxiliary methods (continued).

An around- or primary method can use `next-p` to check whether there is a next method. If the current method is the primary method of an object with no parents, for example, there would be no next method. Since `call-next` yields an error when there is no next method, `next-p` should usually be called to test the waters first. Like `call-next`, `next-p` is defined locally within individual methods.

The new macro `defmeth` is used as follows. If we just want to define the `area` method of the `rectangle` object, we say

```
(setq rectangle (obj))
(defprop height)
(defprop width)
(defmeth (area) rectangle (r)
  (* (height r) (width r)))
```

Now the area of an instance is calculated according to the method of the class:

```
> (let ((myrec (obj rectangle)))
    (setf (height myrec) 2
          (width myrec)  3)
    (area myrec))
6
```

```
(defmacro defmeth ((name &optional (type :primary))
                   obj parms &body body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (unless (meth-p (gethash ',name ,gobj))
         (setf (gethash ',name ,gobj) (make-meth)))
       (setf (,(symb 'meth- type) (gethash ',name ,gobj))
             ,(build-meth name type gobj parms body)))))

(defun build-meth (name type gobj parms body)
  (let ((gargs (gensym)))
    `#'(lambda (&rest ,gargs)
         (labels
           ((call-next ()
              ,(if (or (eq type :primary)
                       (eq type :around))
                   `(cnm ,gobj ',name (cdr ,gargs) ,type)
                   '(error "Illegal call-next.")))
            (next-p ()
              ,(case type
                 (:around
                  `(or (rget ,gobj ',name :around 1)
                       (rget ,gobj ',name :primary)))
                 (:primary
                  `(rget ,gobj ',name :primary 1))
                 (t nil))))
           (apply #'(lambda ,parms ,@body) ,gargs)))))

(defun cnm (obj name args type)
  (case type
    (:around  (let ((ar (rget obj name :around 1)))
                (if ar
                    (apply ar obj args)
                    (run-core-methods obj name args))))
    (:primary (let ((pri (rget obj name :primary 1)))
                (if pri
                    (apply pri obj args)
                    (error "No next method."))))))
```

Figure 25.7: Defining methods.

```
(defmacro undefmeth ((name &optional (type :primary)) obj)
  `(setf (,(symb 'meth- type) (gethash ',name ,obj))
         nil))
```

Figure 25.8: Removing methods.

In a more complicated example, suppose we have defined a `backup` method for the `filesystem` object:

```
(setq filesystem (obj))
(defmeth (backup :before) filesystem (fs)
  (format t "Remember to mount the tape.~%"))
(defmeth (backup) filesystem (fs)
  (format t "Oops, deleted all your files.~%")
  'done)
(defmeth (backup :after) filesystem (fs)
  (format t "Well, that was easy.~%"))
```

The normal sequence of calls will be as follows:

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
DONE
```

Later we want to know how long backups take, so we define the following around-method:

```
(defmeth (backup :around) filesystem (fs)
  (time (call-next)))
```

Now whenever `backup` is called on a child of `filesystem` (unless more specific around-methods intervene) our around-method will be called. It calls the code that would ordinarily run in a call to `backup`, but within a call to `time`. The value returned by `time` will be returned as the value of the call to `backup`:

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
Elapsed Time = .01 seconds
DONE
```

Once we are finished timing the backups, we will want to remove the around-method. That can be done by calling `undefmeth` (Figure 25.8), which takes the same first two arguments as `defmeth`:

```
(undefmeth (backup :around) filesystem)
```

Another thing we might want to alter is an object's list of parents. But after any such change, we should also update the list of ancestors of the object and all its children. So far, we have no way of getting from an object to its children, so we must also add a `children` property.

Figure 25.9 contains code for operating on objects' parents and children. Instead of getting at parents and children via `gethash`, we use the operators `parents` and `children`. The latter is a macro, and therefore transparent to `setf`. The former is a function whose inversion is defined by `defsetf` to be `set-parents`, which does everything needed to maintain consistency in the new doubly-linked world.

To update the ancestors of all the objects in a subtree, `set-parents` calls `maphier`, which is like a `mapc` for inheritance hierarchies. As `mapc` calls a function on every element of a list, `maphier` calls a function on an object and all its descendants. Unless they form a proper tree, the function could get called more than once on some objects. Here this is harmless, because `get-ancestors` does the same thing when called multiple times.

Now we can alter the inheritance hierarchy just by using `setf` on an object's `parents`:

```
> (progn (pop (parents patriotic-scoundrel))
         (serves patriotic-scoundrel))
COUNTRY
T
```

When the hierarchy is modified, affected lists of children and ancestors will be updated automatically. (The `children` are not meant to be manipulated directly, but they could be if we defined a `set-children` analogous to `set-parents`.) The last function in Figure 25.9 is `obj` redefined to use the new code.

As a final improvement to our system, we will make it possible to specify new ways of combining methods. Currently, the only primary method that gets called is the most specific (though it can call others via `call-next`). Instead we might like to be able to combine the results of the primary methods of each of an object's ancestors. For example, suppose that `my-orange` is a child of `orange`, which is a child of `citrus`. If the `props` method returns `(round acidic)` for `citrus`, `(orange sweet)` for `orange`, and `(dented)` for `my-orange`, it would be convenient to be able to make `(props my-orange)` return the *union* of all these values: `(dented orange sweet round acidic)`.

```
(defmacro children (obj)
  `(gethash 'children ,obj))

(defun parents (obj)
  (gethash 'parents obj))

(defun set-parents (obj pars)
  (dolist (p (parents obj))
    (setf (children p)
          (delete obj (children p))))
  (setf (gethash 'parents obj) pars)
  (dolist (p pars)
    (pushnew obj (children p)))
  (maphier #'(lambda (obj)
               (setf (gethash 'ancestors obj)
                     (get-ancestors obj)))
           obj)
  pars)

(defsetf parents set-parents)

(defun maphier (fn obj)
  (funcall fn obj)
  (dolist (c (children obj))
    (maphier fn c)))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (parents obj) parents)
    obj))
```

Figure 25.9: Maintaining parent and child links.

We could have this if we allowed methods to apply some function to the values of all the primary methods, instead of just returning the value of the most specific. Figure 25.10 contains a macro which allows us to define the way methods are combined, and a new version of `run-core-methods` which can perform method combination.

We define the form of combination for a method via `defcomb`, which takes a method name and a second argument describing the desired combination. Or-

```
(defmacro defcomb (name op)
  `(progn
     (defprop ,name t)
     (setf (get ',name 'mcombine)
           ,(case op
              (:standard nil)
              (:progn '#'(lambda (&rest args)
                             (car (last args))))
              (t op)))))

(defun run-core-methods (obj name args &optional pri)
  (let ((comb (get name 'mcombine)))
    (if comb
        (if (symbolp comb)
            (funcall (case comb (:and #'comb-and)
                                (:or  #'comb-or))
                     obj name args (ancestors obj))
            (comb-normal comb obj name args))
        (multiple-value-prog1
          (progn (run-befores obj name args)
                 (apply (or pri (rget obj name :primary))
                        obj args))
          (run-afters obj name args)))))

(defun comb-normal (comb obj name args)
  (apply comb
         (mapcan #'(lambda (a)
                     (let* ((pm (meth- primary
                                       (gethash name a)))
                            (val (if pm
                                     (apply pm obj args))))
                       (if val (list val))))
                 (ancestors obj))))
```

Figure 25.10: Method combination.

dinarily this second argument should be a function. However, it can also be one of `:progn`, `:and`, `:or`, or `:standard`. With the former three, primary methods will be combined as though according to the corresponding operator, while `:standard` indicates that we want the traditional way of running methods.

```
(defun comb-and (obj name args ancs &optional (last t))
  (if (null ancs)
      last
      (let ((pm (meth- primary (gethash name (car ancs)))))
        (if pm
            (let ((new (apply pm obj args)))
              (and new
                   (comb-and obj name args (cdr ancs) new)))
            (comb-and obj name args (cdr ancs) last)))))

(defun comb-or (obj name args ancs)
  (and ancs
       (let ((pm (meth- primary (gethash name (car ancs)))))
         (or (and pm (apply pm obj args))
             (comb-or obj name args (cdr ancs))))))
```

Figure 25.11: Method combination (continued).

The central function in Figure 25.10 is the new `run-core-methods`. If the method being called has no `mcombine` property, then the method call proceeds as before. Otherwise the `mcombine` of the method is either a function (like `+`) or a keyword (like `:or`). In the former case, the function is just applied to a list of the values returned by all the primary methods.[1] In the latter, we use the function associated with the keyword to iterate over the primary methods.

The operators `and` and `or` have to be treated specially, as in Figure 25.11. They get special treatment not just because they are special forms, but because they short-circuit evaluation:

```
> (or 1 (princ "wahoo"))
1
```

Here nothing is printed because the `or` returns as soon as it sees a non-nil argument. Similarly, a primary method subject to `or` combination should never get called if a more specific method returns true. To provide such short-circuiting for `and` and `or`, we use the distinct functions `comb-and` and `comb-or`.

To implement our previous example, we would write:

```
(setq citrus (obj))
(setq orange (obj citrus))
```

[1] A more sophisticated version of this code could use `reduce` to avoid consing here.

```
(setq my-orange (obj orange))

(defmeth (props) citrus (c) '(round acidic))
(defmeth (props) orange (o) '(orange sweet))
(defmeth (props) my-orange (m) '(dented))

(defcomb props #'(lambda (&rest args) (reduce #'union args)))
```

after which `props` would return the union of all the primary method values:[2]

```
> (props my-orange)
(DENTED ORANGE SWEET ROUND ACIDIC)
```

Incidentally, this example suggests a choice that you only have when doing object-oriented programming in Lisp: whether to store information in slots or methods.

Afterward, if we wanted the `props` method to return to the default behavior, we just set the method combination back to standard:

```
> (defcomb props :standard)
NIL
> (props my-orange)
(DENTED)
```

Note that before- and after-methods only run in standard method combination. However, around-methods work the same as before.

The program presented in this section is intended as a model, not as a real foundation for object-oriented programming. It was written for brevity rather than efficiency. However, it is at least a working model, and so could be used for experiments and prototypes. If you do want to use the program for such purposes, one minor change would make it much more efficient: don't calculate or store ancestor lists for objects with only one parent.

25.3 Classes and Instances

The program in the previous section was written to resemble CLOS as closely as such a small program could. By understanding it we are already a fair way towards understanding CLOS. In the next few sections we will examine CLOS itself.

In our sketch, we made no syntactic distinction between classes and instances, or between slots and methods. In CLOS, we use the `defclass` macro to define a class, and we declare the slots in a list at the same time:

---

[2] Since the combination function for `props` calls `union`, the list elements will not necessarily be in this order.

```
(defclass circle ()
  (radius center))
```

This expression says that the `circle` class has no superclasses, and two slots, `radius` and `center`. We can make an instance of the circle class by saying:

```
(make-instance 'circle)
```

Unfortunately, we have defined no way of referring to the slots of a `circle`, so any instance we make is going to be rather inert. To get at a slot we define an accessor function for it:

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

Now if we make an instance of a circle, we can set its `radius` and `center` slots by using `setf` with the corresponding accessor functions:

```
> (setf (circle-radius (make-instance 'circle)) 2)
2
```

We can do this kind of initialization right in the call to `make-instance` if we define the slots to allow it:

```
(defclass circle ()
  ((radius :accessor circle-radius :initarg :radius)
   (center :accessor circle-center :initarg :center)))
```

The `:initarg` keyword in a slot definition says that the following argument should become a keyword parameter in `make-instance`. The value of the keyword parameter will become the initial value of the slot:

```
> (circle-radius (make-instance 'circle
                   :radius 2
                   :center '(0 . 0)))
2
```

By declaring an `:initform`, we can also define slots which initialize themselves. The `visible` slot of the `shape` class

```
(defclass shape ()
  ((color   :accessor shape-color   :initarg :color)
   (visible :accessor shape-visible :initarg :visible
            :initform t)))
```

will be set to `t` by default:

```
> (shape-visible (make-instance 'shape))
T
```

If a slot has both an initarg and an initform, the initarg takes precedence when it is specified:

```
> (shape-visible (make-instance 'shape :visible nil))
NIL
```

Slots are inherited by instances and subclasses. If a class has more than one superclass, it inherits the union of their slots. So if we define the class `screen-circle` to be a subclass of both `circle` and `shape`,

```
(defclass screen-circle (circle shape)
  nil)
```

then instances of `screen-circle` will have four slots, two inherited from each grandparent. Note that a class does not have to create any new slots of its own; this class exists just to provide something instantiable that inherits from both `circle` and `shape`.

The accessors and initargs work for instances of `screen-circle` just as they would for instances of `circle` or `shape`:

```
> (shape-color (make-instance 'screen-circle
                              :color 'red :radius 3))
RED
```

We can cause every `screen-circle` to have some default initial `color` by specifying an initform for this slot in the `defclass`:

```
(defclass screen-circle (circle shape)
  ((color :initform 'purple)))
```

Now instances of `screen-circle` will be purple by default,

```
> (shape-color (make-instance 'screen-circle))
PURPLE
```

though it is still possible to initialize the slot otherwise by giving an explicit `:color` initarg.

In our sketch of object-oriented programming, instances inherited values directly from the slots in their parent classes. In CLOS, instances do not have slots in the same way that classes do. We define an inherited default for instances by defining an initform in the parent class. In a way, this is more flexible, because as well as being a constant, an initform can be an expression that returns a different value each time it is evaluated:

```
(defclass random-dot ()
  ((x :accessor dot-x :initform (random 100))
   (y :accessor dot-y :initform (random 100))))
```

Each time we make an instance of a `random-dot` its `x`- and `y`-position will be a random integer between `0` and `99`:

```
> (mapcar #'(lambda (name)
              (let ((rd (make-instance 'random-dot)))
                (list name (dot-x rd) (dot-y rd))))
          '(first second third))
((FIRST 25 8) (SECOND 26 15) (THIRD 75 59))
```

In our sketch, we also made no distinction between slots whose values were to vary from instance to instance, and those which were to be constant across the whole class. In CLOS we can specify that some slots are to be *shared*—that is, their value is the same for every instance. We do this by declaring the slot to have `:allocation :class`. (The alternative is for a slot to have `:allocation :instance`, but since this is the default there is no need to say so explicitly.) For example, if all owls are nocturnal, then we can make the `nocturnal` slot of the `owl` class a shared slot, and give it the initial value `t`:

```
(defclass owl ()
  ((nocturnal :accessor owl-nocturnal
              :initform t
              :allocation :class)))
```

Now every instance of the `owl` class will inherit this slot:

```
> (owl-nocturnal (make-instance 'owl))
T
```

If we change the "local" value of this slot in an instance, we are actually altering the value stored in the class:

```
> (setf (owl-nocturnal (make-instance 'owl)) 'maybe)
MAYBE
> (owl-nocturnal (make-instance 'owl))
MAYBE
```

This could cause some confusion, so we might like to make such a slot read-only. When we define an accessor function for a slot, we create a way of both reading and writing the slot's value. If we want the value to be readable but not writable, we can do it by giving the slot just a reader function, instead of a full-fledged accessor function:

```
(defclass owl ()
  ((nocturnal :reader owl-nocturnal
              :initform t
              :allocation :class)))
```

Now attempts to alter the `nocturnal` slot of an instance will generate an error:

```
> (setf (owl-nocturnal (make-instance 'owl)) nil)
>>Error: The function (SETF OWL-NOCTURNAL) is undefined.
```

25.4 Methods

Our sketch emphasized the similarity between slots and methods in a language which provides lexical closures. In our program, a primary method was stored and inherited in the same way as a slot value. The only difference between a slot and a method was that defining a name as a slot by

```
(defprop area)
```

made `area` a function which would simply retrieve and return a value, while defining it as a method by

```
(defprop area t)
```

made `area` a function which would, after retrieving a value, funcall it on its arguments.

In CLOS the functional units are still called methods, and it is possible to define them so that they each seem to be a property of some class. Here we define an `area` method for the `circle` class:

```
(defmethod area ((c circle))
  (* pi (expt (circle-radius c) 2)))
```

The parameter list for this method says that it is a function of one argument which applies to instances of the `circle` class.

We invoke this method like a function, just as in our sketch:

```
> (area (make-instance 'circle :radius 1))
3.14...
```

We can also define methods that take additional arguments:

```
(defmethod move ((c circle) dx dy)
  (incf (car (circle-center c)) dx)
  (incf (cdr (circle-center c)) dy)
  (circle-center c))
```

If we call this method on an instance of `circle`, its center will be shifted by ⟨`dx`,`dy`⟩:

```
> (move (make-instance 'circle :center '(1 . 1)) 2 3)
(3 . 4)
```

The value returned by the method reflects the circle's new position.

As in our sketch, if there is a method for the class of an instance, and for superclasses of that class, the most specific one runs. So if `unit-circle` is a subclass of `circle`, with the following `area` method

```
(defmethod area ((c unit-circle)) pi)
```

then this method, rather than the more general one, will run when we call `area` on an instance of `unit-circle`.

When a class has multiple superclasses, their precedence runs left to right. By defining the class `patriotic-scoundrel` as follows

```
(defclass scoundrel nil nil)
(defclass patriot nil nil)
(defclass patriotic-scoundrel (scoundrel patriot) nil)
```

we specify that patriotic scoundrels are scoundrels first and patriots second. When there is an applicable method for both superclasses,

```
(defmethod self-or-country? ((s scoundrel))
  'self)

(defmethod self-or-country? ((p patriot))
  'country)
```

the method of the `scoundrel` class will run:

```
> (self-or-country? (make-instance 'patriotic-scoundrel))
SELF
```

The examples so far maintain the illusion that CLOS methods are methods *of* some object. In fact, they are something more general. In the parameter list of the `move` method, the element `(c circle)` is called a *specialized* parameter; it says that this method applies when the first argument to `move` is an instance of the `circle` class. In a CLOS method, *more than one parameter can be specialized.* The following method has two specialized and one optional unspecialized parameter:

```
(defmethod combine ((ic ice-cream) (top topping)
                    &optional (where :here))
  (append (list (name ic) 'ice-cream)
          (list 'with (name top) 'topping)
          (list 'in 'a
                (case where
                  (:here  'glass)
                  (:to-go 'styrofoam))
                'dish)))
```

It is invoked when the first two arguments to `combine` are instances of `ice-cream` and `topping`, respectively. If we define some minimal classes to instantiate

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) nil)
(defclass topping (stuff) nil)
```

then we can define and run this method:

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'olive)
           :here)
(FIG ICE-CREAM WITH OLIVE TOPPING IN A GLASS DISH)
```

When methods specialize more than one of their parameters, it is difficult to continue to regard them as properties of classes. Does our `combine` method belong to the `ice-cream` class or the `topping` class? In CLOS, the model of objects responding to messages simply evaporates. This model seems natural so long as we invoke methods by saying something like:

```
(tell obj 'move 2 3)
```

Then we are clearly invoking the `move` method of `obj`. But once we drop this syntax in favor of a functional equivalent:

```
(move obj 2 3)
```

then we have to define `move` so that it *dispatches* on its first argument—that is, looks at the type of the first argument and calls the appropriate method.

Once we have taken this step, the question arises: why only allow dispatching on the first argument? CLOS answers: why indeed? In CLOS, methods can specialize any number of their parameters—and not just on user-defined classes, but on Common Lisp types,[3] and even on individual objects. Here is a `combine` method that applies to strings:

[3]Or more precisely, on the type-like classes that CLOS defines in parallel with the Common Lisp type hierarchy.

```
(defmethod combine ((s1 string) (s2 string) &optional int?)
  (let ((str (concatenate 'string s1 s2)))
    (if int? (intern str) str)))
```

Which means not only that methods are no longer properties of classes, but that we can use methods without defining classes at all.

```
> (combine "I am not a " "cook.")
"I am not a cook."
```

Here the second parameter is specialized on the symbol `palindrome`:

```
(defmethod combine ((s1 sequence) (x (eql 'palindrome))
                    &optional (length :odd))
  (concatenate (type-of s1)
               s1
               (subseq (reverse s1)
                       (case length (:odd 1) (:even 0)))))
```

This particular method makes palindromes of any kind of sequence elements:[4]

```
> (combine '(able was i ere) 'palindrome)
(ABLE WAS I ERE I WAS ABLE)
```

At this point we no longer have object-oriented programming, but something more general. CLOS is designed with the understanding that beneath methods there is this concept of dispatch, which can be done on more than one argument, and can be based on more than an argument's class. When methods are built upon this more general notion, they become independent of individual classes. Instead of adhering conceptually to classes, methods now adhere to other methods with the same name. In CLOS such a clump of methods is called a *generic function.* All our `combine` methods implicitly define the generic function `combine`.

We can define generic functions explicitly with the `defgeneric` macro. It is not necessary to call `defgeneric` to define a generic function, but it can be a convenient place to put documentation, or some sort of safety-net for errors. Here we do both:

```
(defgeneric combine (x y &optional z)
  (:method (x y &optional z)
     "I can't combine these arguments.")
  (:documentation "Combines things."))
```

[4]In one (otherwise excellent) Common Lisp implementation, `concatenate` will not accept `cons` as its first argument, so this call will not work.

Since the method given here for `combine` doesn't specialize any of its arguments, it will be the one called in the event no other method is applicable.

```
> (combine #'expt "chocolate")
"I can't combine these arguments."
```

Before, this call would have generated an error.

Generic functions impose one restriction that we don't have when methods are properties of objects: when all methods of the same name get joined into one generic function, their parameter lists must agree. That's why all our `combine` methods had an additional optional parameter. After defining the first `combine` method to take up to three arguments, it would have caused an error if we attempted to define another which only took two.

CLOS requires that the parameter lists of all methods with the same name be *congruent.* Two parameter lists are congruent if they have the same number of required parameters, the same number of optional parameters, and compatible use of `&rest` and `&key`. The actual keyword parameters accepted by different methods need not be the same, but `defgeneric` can insist that all its methods accept a certain minimal set. The following pairs of parameter lists are all congruent:

```
(x)              (a)
(x &optional y) (a &optional b)
(x y &rest z)   (a b &rest c)
(x y &rest z)   (a b &key c d)
```

and the following pairs are not:

```
(x)              (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y)    (a)
```

Redefining methods is just like redefining functions. Since only required parameters can be specialized, each method is uniquely identified by its generic function and the types of its required parameters. If we define another method with the same specializations, it overwrites the original one. So by saying:

```
(defmethod combine ((x string) (y string)
                    &optional ignore)
  (concatenate 'string x " + " y))
```

we redefine what `combine` does when its first two arguments are strings.

Unfortunately, if instead of redefining a method we want to remove it, there is no built-in converse of `defmethod`. Fortunately, this is Lisp, so we can write

```
(defmacro undefmethod (name &rest args)
  (if (consp (car args))
      (udm name nil (car args))
      (udm name (list (car args)) (cadr args))))

(defun udm (name qual specs)
  (let ((classes (mapcar #'(lambda (s)
                              `(find-class ',s))
                          specs)))
    `(remove-method (symbol-function ',name)
                    (find-method (symbol-function ',name)
                                 ',qual
                                 (list ,@classes)))))
```

Figure 25.12: Macro for removing methods.

one. The details of how to remove a method by hand are summarized in the implementation of `undefmethod` in Figure 25.12. We use this macro by giving arguments similar to those we would give to `defmethod`, except that instead of giving a whole parameter list as the second or third argument, we give just the class-names of the required parameters. So to remove the `combine` method for two strings, we say:

```
(undefmethod combine (string string))
```

Unspecialized arguments are implicitly of class `t`, so if we had defined a method with required but unspecialized parameters:

```
(defmethod combine ((fn function) x &optional y)
  (funcall fn x y))
```

we could get rid of it by saying

```
(undefmethod combine (function t))
```

If we want to remove a whole generic function, we can do it the same way we would remove the definition of any function, by calling `fmakunbound`:

```
(fmakunbound 'combine)
```

25.5 Auxiliary Methods and Combination

Auxiliary methods worked in our sketch basically as they do in CLOS. So far we have seen only primary methods, but we can also have before-, after- and around-methods. Such auxiliary methods are defined by putting a qualifying keyword after the method name in the call to `defmethod`. If we define a primary `speak` method for the `speaker` class as follows:

```
(defclass speaker nil nil)

(defmethod speak ((s speaker) string)
  (format t "~A" string))
```

Then calling `speak` with an instance of `speaker` just prints the second argument:

```
> (speak (make-instance 'speaker)
         "life is not what it used to be")
life is not what it used to be
NIL
```

By defining a subclass `intellectual` which wraps before- and after-methods around the primary `speak` method,

```
(defclass intellectual (speaker) nil)

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

we can create a subclass of speakers which always have the last (and the first) word:

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps life is not what it used to be in some sense
NIL
```

In standard method combination, the methods are called as described in our sketch: all the before-methods, most specific first, then the most specific primary method, then all the after-methods, most specific last. So if we define before- or after-methods for the `speaker` superclass,

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

they will get called in the middle of the sandwich:

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps I think life is not what it used to be in some sense
NIL
```

Regardless of what before- or after-methods get called, the value returned by the generic function is the value of the most specific primary method—in this case, the `nil` returned by `format`.

This changes if there are around-methods. If one of the classes in an object's family tree has an around-method—or more precisely, if there is an around-method specialized for the arguments passed to the generic function—the around-method will get called first, and the rest of the methods will only run if the around-method decides to let them. As in our sketch, an around- or primary method can invoke the next method by calling a function: the function we defined as `call-next` is in CLOS called `call-next-method`. There is also a `next-method-p`, analogous to our `next-p`. With around-methods we can define another subclass of `speaker` which is more circumspect:

```
(defclass courtier (speaker) nil)

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
  (if (eq (read) 'yes)
      (if (next-method-p) (call-next-method))
      (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

When the first argument to `speak` is an instance of the `courtier` class, the courtier's tongue is now guarded by the around-method:

```
> (speak (make-instance 'courtier) "kings will last")
Does the King believe that kings will last? yes
I think kings will last
BOW
> (speak (make-instance 'courtier) "the world is round")
Does the King believe that the world is round? no
Indeed, it is a preposterous idea.
BOW
```

Note that, unlike before- and after-methods, the value returned by the around-method is returned as the value of the generic function.

Generally, methods are run as in this outline, which is reprinted from Section 25.2:

1. The most specific around-method, if there is one.
2. Otherwise, in order:
   (a) All before-methods, from most specific to least specific.
   (b) The most specific primary method.
   (c) All after-methods, from least specific to most specific.

This way of combining methods is called standard method combination. As in our sketch, it is possible to define methods which are combined in other ways: for example, for a generic function to return the sum of all the applicable primary methods.

In our program, we specified how to combine methods by calling `defcomb`. By default, methods were combined as in the outline above, but by saying, for example,

```
(defcomb price #'+)
```

we could cause the function `price` to return the sum of all the applicable primary methods.

In CLOS this is called *operator* method combination. As in our program, such method combination can be understood as if it resulted in the evaluation of a Lisp expression whose first element was some operator, and whose arguments were calls to the applicable primary methods, in order of specificity. If we defined the `price` generic function to combine values with `+`, and there were no applicable around-methods, it would behave as though it were defined:

```
(defun price (&rest args)
  (+ (apply ⟨most specific primary method⟩ args)
     ⋮
     (apply ⟨least specific primary method⟩ args)))
```

If there are applicable around-methods, they take precedence, just as in standard method combination. Under operator method combination, an around-method can still call the next method via `call-next-method`. However, primary methods can no longer use `call-next-method`. (This is a difference from our sketch, where we left `call-next` available to such methods.)

In CLOS, we can specify the type of method combination to be used by a generic function by giving the optional `:method-combination` argument to `defgeneric`:

```
(defgeneric price (x)
  (:method-combination +))
```

Now the `price` method will use + method combination. If we define some classes with prices,

```
(defclass jacket nil nil)
(defclass trousers nil nil)
(defclass suit (jacket trousers) nil)

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

then when we ask for the price of an instance of `suit`, we get the sum of the applicable `price` methods:

```
> (price (make-instance 'suit))
550
```

The following symbols can be used as the second argument to `defmethod` or in the `:method-combination` option to `defgeneric`:

```
+   and   append   list   max   min   nconc   or   progn
```

By calling `define-method-combination` you can define other kinds of method combination; see CLTL2, p. 830.

Once you specify the method combination a generic function should use, all methods for that function must use the same kind. Now it would cause an error if we tried to use another operator (or `:before` or `:after`) as the second argument in a `defmethod` for `price`. If we do want to change the method combination of `price` we must remove the whole generic function by calling `fmakunbound`.

25.6 CLOS and Lisp

CLOS makes a good example of an embedded language. This kind of program usually brings two rewards:

1. Embedded languages can be conceptually well-integrated with their environment, so that within the embedded language we can continue to think of programs in much the same terms.

2. Embedded languages can be powerful, because they take advantage of all the things that the base language already knows how to do.

CLOS wins on both counts. It is very well-integrated with Lisp, and it makes good use of the abstractions that Lisp has already. Indeed, we can often see Lisp through CLOS, the way we can see the shapes of objects through a sheet draped over them.

It is no accident that we usually speak to CLOS through a layer of macros. Macros do transformation, and CLOS is essentially a program which takes programs built out of object-oriented abstractions, and translates them into programs built out of Lisp abstractions.

As the first two sections suggested, the abstractions of object-oriented programming map so neatly onto those of Lisp that one could almost call the former a special case of the latter. The objects of object-oriented programming can easily be implemented as Lisp objects, and their methods as lexical closures. By taking advantage of such isomorphisms, we were able to provide a rudimentary form of object-oriented programming in just a few lines of code, and a sketch of CLOS in a few pages.

CLOS is a great deal larger and more powerful than our sketch, but not so large as to disguise its roots as an embedded language. Take `defmethod` as an example. Though CLTL2 does not mention it explicitly, CLOS methods have all the power of lexical closures. If we define several methods within the scope of some variable,

```
(let ((transactions 0))
  (defmethod withdraw ((a account) amt)
    (incf transactions)
    (decf (balance a) amt))
  (defmethod deposit ((a account) amt)
    (incf transactions)
    (incf (balance a) amt))
  (defun transactions ()
    transactions))
```

then at runtime they will share access to the variable, just like closures. Methods can do this because, underneath the syntax, they *are* closures. In the expansion of a `defmethod`, its body appears intact in the body of a sharp-quoted lambda-expression.

Section 7.6 suggested that it was easier to conceive of how macros work than what they mean. Likewise, the secret to understanding CLOS is to understand how it maps onto the fundamental abstractions of Lisp.

25.7 When to Object

The object-oriented style provides several distinct benefits. Different programs need these benefits to varying degrees. At one end of the continuum there are programs—simulations, for example—which are most naturally expressed in the abstractions of object-oriented programming. At the other end are programs written in the object-oriented style mainly to make them extensible.

Extensibility is indeed one of the great benefits of the object-oriented style. Instead of being a single monolithic blob of code, a program is written in small pieces, each labelled with its purpose. So later when someone else wants to modify the program, it will be easy to find the part that needs to be changed. If we want to change the way that objects of type `ob` are displayed on the screen, we change the `display` method of the `ob` class. If we want to make a new class of objects like `ob`s but different in a few respects, we can create a subclass of `ob`; in the subclass, we change the properties we want, and all the rest will be inherited by default from the `ob` class. And if we just want to make a single `ob` which behaves differently from the rest, we can create a new child of `ob` and modify the child's properties directly. If the program was written carefully to begin with, we can make all these types of modifications without even looking at the rest of the code. From this point of view, an object-oriented program is a program organized like a table: we can change it quickly and safely by looking up the appropriate entry.

Extensibility demands the least from the object-oriented style. In fact, it demands so little that an extensible program might not need to be object-oriented at all. If the preceding chapters have shown anything, they have shown that Lisp programs do not have to be monolithic blobs of code. Lisp offers a whole range of options for extensibility. For example, you could quite literally have a program organized like a table: a program which consisted of a set of closures stored in an array.

If it's extensibility you need, you don't have to choose between an "object-oriented" and a "traditional" program. You can give a Lisp program exactly the degree of extensibility it needs, often without resorting to object-oriented techniques. A slot in a class is a global variable. And just as it is inelegant to use a global variable where you could use a parameter, it could be inelegant to build a world of classes and instances when you could do the same thing with less effort in plain Lisp. With the addition of CLOS, Common Lisp has become the most powerful object-oriented language in widespread use. Ironically, it is also the language in which object-oriented programming is least necessary.

Appendix: Packages

Packages are Common Lisp's way of grouping code into modules. Early dialects of Lisp contained a symbol-table, called the *oblist*, which listed all the symbols read so far by the system. Through a symbol's entry on the oblist, the system had access to things like its value and its property list. A symbol listed in the oblist was said to be *interned.*

Recent dialects of Lisp have split the concept of the oblist into multiple *packages.* Now a symbol is not merely interned, but interned in a particular package. Packages support modularity because symbols interned in one package are only accessible in other packages (except by cheating) if they are explicitly declared to be so.

A package is a kind of Lisp object. The current package is always stored in the global variable `*package*`. When Common Lisp starts up, the current package will be the user package: either `user` (in CLTL1 implementations), or `common-lisp-user` (in CLTL2 implementations).

Packages are usually identified by their names, which are strings. To find the name of the current package, try:

```
> (package-name *package*)
"COMMON-LISP-USER"
```

Usually a symbol is interned in the package that was current at the time it was read. To find the package in which a symbol is interned, we can use `symbol-package`:

```
> (symbol-package 'foo)
#<Package "COMMON-LISP-USER" 4CD15E>
```

The return value here is the actual package object. For future use, let's give `foo` a value:

```
> (setq foo 99)
99
```

By calling `in-package` we can switch to a new package, creating it if necessary:[1]

```
> (in-package 'mine :use 'common-lisp)
#<Package "MINE" 63390E>
```

At this point there should be eerie music, because we are in a different world: `foo` here is not what it used to be.

```
MINE> foo
>>Error: FOO has no global value.
```

Why did this happen? Because the `foo` we set to 99 above is a distinct symbol from `foo` here in `mine`.[2] To refer to the original `foo` from outside the user package, we must prefix the package name and two colons:

```
MINE> common-lisp-user::foo
99
```

So different symbols with the same print-name can coexist in different packages. There can be one `foo` in package `common-lisp-user` and another `foo` in package `mine`, and they will be distinct symbols. In fact, that's partly the point of packages: if you're writing your program in a separate package, you can choose names for your functions and variables without worrying that someone will use the same name for something else. Even if they use the same name, it won't be the same symbol.

Packages also provide a means of information-hiding. Programs must refer to functions and variables by their names. If you don't make a given name available outside your package, it becomes unlikely that code in another package will be able to use or modify what it refers to.

In programs it's usually bad style to use package prefixes with double colons. By doing so you are violating the modularity that packages are supposed to provide. If you have to use a double colon to refer to a symbol, it's because someone didn't want you to.

[1] In older implementations of Common Lisp, omit the `:use` argument.

[2] Some implementations of Common Lisp print the package name before the toplevel prompt whenever we are not in the user package. This is not required, but it is a nice touch.

Usually one should only refer to symbols which have been *exported.* By exporting a symbol from the package in which it is interned, we cause it to be visible to other packages. To export a symbol we call (you guessed it) `export`:

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
> (setq bar 5)
5
```

Now when we return to `mine`, we can refer to `bar` with only a single colon, because it is a publicly available name:

```
> (in-package 'mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

By *importing* `bar` into `mine` we can go one step further, and make `mine` actually share the symbol `bar` with the user package:

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

After importing `bar` we can refer to it without any package qualifier at all. The two packages now share the same symbol; there can't be a distinct `mine:bar`.

What if there already was one? In that case, the call to `import` would have caused an error, as we see if we try to import `foo`:

```
MINE> (import 'common-lisp-user::foo)
>>Error: FOO is already present in MINE.
```

Before, when we tried unsuccessfully to evaluate `foo` in `mine`, we thereby caused a symbol `foo` to be interned there. It had no global value and therefore generated an error, but the interning happened simply as a consequence of typing its name. So now when we try to import `foo` into `mine`, there is already a symbol there with the same name.

We can also import symbols *en masse* by defining one package to *use* another:

```
MINE> (use-package 'common-lisp-user)
T
```

Now all symbols exported by the user package will automatically be imported by `mine`. (If `foo` had been exported by the user package, this call would also have generated an error.)

As of CLTL2, the package containing the names of built-in operators and variables is called `common-lisp` instead of `lisp`, and new packages no longer use it by default. Since we used this package in the call to `in-package` which created `mine`, all of Common Lisp's names will be visible here:

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

You're practically compelled to make any new package use `common-lisp` (or some other package containing Lisp operators). Otherwise you wouldn't even be able to get out of the new package.

As with compilation, operations on packages are not usually done at the toplevel like this. More often the calls are contained in source files. Generally it will suffice to begin a file with an `in-package` and a `defpackage`. (The `defpackage` macro is new in CLTL2, but some older implementations provide it.) Here is what you might put at the top of a file containing a distinct package of code:

```
(in-package 'my-application :use 'common-lisp)

(defpackage my-application
            (:use common-lisp my-utilities)
            (:nicknames app)
            (:export win lose draw))
```

This will cause the code in the file—or more precisely, the names in the file—to be in the package `my-application`. As well as `common-lisp`, this package uses `my-utilities`, so any symbols exported thence can appear without any package prefix in the file.

The `my-application` package itself exports just three symbols: `win`, `lose`, and `draw`. Since the call to `in-package` gave `my-application` the nickname `app`, code in other packages will be able to refer to them as e.g. `app:win`.

The kind of modularity provided by packages is actually a bit odd. We have modules not of objects, but of names. Every package that uses `common-lisp` imports the name `cons`, because `common-lisp` includes a function with that name. But in consequence a variable called `cons` would also be visible every package that used `common-lisp`. And the same thing goes for Common Lisp's other name-spaces. If packages are confusing, this is the main reason why; they're not based on objects, but on names.

Things having to do with packages tend to happen at read-time, not runtime, which can lead to some confusion. The second expression we typed:

```
(symbol-package 'foo)
```

returned the value it did because *reading the query created the answer.* To evaluate this expression, Lisp had to read it, which meant interning `foo`.

As another example, consider this exchange, which appeared above:

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
```

Usually two expressions typed into the toplevel are equivalent to the same two expressions enclosed within a single `progn`. Not in this case. If we try saying

```
MINE> (progn (in-package 'common-lisp-user)
             (export 'bar))
>>Error: MINE::BAR is not accessible in COMMON-LISP-USER.
```

we get an error instead. This happens because the whole `progn` expression is processed by `read` before being evaluated. When `read` is called, the current package is `mine`, so `bar` is taken to be `mine:bar`. It is as if we had asked to export this symbol, instead of `common-lisp-user:bar`, from the user package.

The way packages are defined makes it a nuisance to write programs which use symbols as data. For example, if we define `noise` as follows:

```
(in-package 'other :use 'common-lisp)
(defpackage other
            (:use common-lisp)
            (:export noise))

(defun noise (animal)
  (case animal
    (dog 'woof)
    (cat 'meow)
    (pig 'oink)))
```

then if we call `noise` from another package with an unqualified symbol as an argument, it will usually fall off the end of the `case` clauses and return `nil`:

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise 'pig)
NIL
```

That's because what we passed as an argument was `common-lisp-user:pig` (no offense intended), while the `case` key is `other:pig`. To make `noise` work as one would expect, we would have to export all six symbols used within it, and import them into any package from which we intended to call `noise`.

In this case, we could evade the problem by using keywords instead of ordinary symbols. If `noise` had been defined

```
(defun noise (animal)
  (case animal
    (:dog :woof)
    (:cat :meow)
    (:pig :oink)))
```

then we could safely call it from any package:

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise :pig)
:OINK
```

Keywords are like gold: universal and self-evaluating. They are visible everywhere, and they never have to be quoted. A symbol-driven function like `defanaph` (page 223) should nearly always be written to use keywords.

Packages are a rich source of confusion. This introduction to the subject has barely scratched the surface. For all the details, see CLTL2, Chapter 11.

Notes

This section is also intended as a bibliography. All the books and papers listed here should be considered recommended reading.

v Foderaro, John K. Introduction to the Special Lisp Section. *CACM* 34, 9 (September 1991), p. 27.

viii The final Prolog implementation is 94 lines of code. It uses 90 lines of utilities from previous chapters. The ATN compiler adds 33 lines, for a total of 217. Since Lisp has no formal notion of a line, there is a large margin for error when measuring the length of a Lisp program in lines.

ix Steele, Guy L., Jr. *Common Lisp: the Language,* 2nd Edition. Digital Press, Bedford (MA), 1990.

5 Brooks, Frederick P. *The Mythical Man-Month.* Addison-Wesley, Reading (MA), 1975, p. 16.

18 Abelson, Harold, and Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, 1985.

21 More precisely, we cannot define a recursive function with a single lambda-expression. We can, however, generate a recursive function by writing a function to take *itself* as an additional argument,

```
(setq fact
      #'(lambda (f n)
          (if (= n 0)
              1
              (* n (funcall f f (- n 1))))))
```

and then passing it to a function that will return a closure in which original function is called on itself:

```
(defun recurser (fn)
  #'(lambda (&rest args)
      (apply fn fn args)))
```

Passing `fact` to this function yields a regular factorial function,

```
> (funcall (recurser fact) 8)
40320
```

which could have been expressed directly as:

```
((lambda (f) #'(lambda (n) (funcall f f n)))
 #'(lambda (f n)
     (if (= n 0)
         1
         (* n (funcall f f (- n 1))))))
```

Many Common Lisp users will find `labels` or `alambda` more convenient.

23 Gabriel, Richard P. Performance and Standardization. *Proceedings of the First International Workshop on Lisp Evolution and Standardization,* 1988, p. 60.

Testing `triangle` in one implementation, Gabriel found that "even when the C compiler is provided with hand-generated register allocation information, the Lisp code is 17% faster than an iterative C version of this function." His paper mentions several other programs which ran faster in Lisp than in C, including one that was 42% faster.

24 If you wanted to compile all the named functions currently loaded, you could do it by calling `compall`:

```
(defun compall ()
  (do-symbols (s)
    (when (fboundp s)
      (unless (compiled-function-p (symbol-function s))
        (print s)
        (compile s)))))
```

This function also prints the name of each function as it is compiled.

26 You may be able to see whether `inline` declarations are being obeyed by calling `(disassemble 'foo)`, which displays some representation of the object code of function `foo`. This is also one way to check whether tail-recursion optimization is being done.

31 One could imagine `nreverse` defined as:

```
(defun our-nreverse (lst)
  (if (null (cdr lst))
      lst
      (prog1 (nr2 lst)
             (setf (cdr lst) nil))))
```

```
(defun nr2 (lst)
  (let ((c (cdr lst)))
    (prog1 (if (null (cdr c))
               c
               (nr2 c))
           (setf (cdr c) lst))))
```

43 Good design always puts a premium on economy, but there is an additional reason that programs should be dense. When a program is dense, you can see more of it at once.

People know intuitively that design is easier when one has a broad view of one's work. This is why easel painters use long-handled brushes, and often step back from their work. This is why generals position themselves on high ground, even if they are thereby exposed to enemy fire. And it is why programmers spend a lot of money to look at their programs on large displays instead of small ones.

Dense programs make the most of one's field of vision. A general cannot shrink a battle to fit on a table-top, but Lisp allows you to perform corresponding feats of abstraction in programs. And the more you can see of your program at once, the more likely it is to turn out as a unified whole.

This is not to say that one should make one's programs shorter at any cost. If you take all the newlines out of a function, you can fit it on one line, but this does not make it easier to read. Dense code means code which has been made smaller by abstraction, not text-editing.

Imagine how hard it would be to program if you had to look at your code on a display half the size of the one you're used to. Making your code twice as dense will make programming that much easier.

44 Steele, Guy L., Jr. Debunking the "Expensive Procedure Call" Myth or, Procedural Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. *Proceedings of the National Conference of the ACM,* 1977, p. 157.

48 For reference, here are simpler definitions of some of the functions in Figures 4.2 and 4.3. All are substantially (at least 10%) slower:

```
(defun filter (fn lst)
  (delete nil (mapcar fn lst)))

(defun filter (fn lst)
  (mapcan #'(lambda (x)
              (let ((val (funcall fn x)))
                (if val (list val))))
          lst))

(defun group (source n)
  (if (endp source)
      nil
      (let ((rest (nthcdr n source)))
        (cons (if (consp rest) (subseq source 0 n) source)
              (group rest n)))))
```

```
(defun flatten (x)
  (mapcan #'(lambda (x)
              (if (atom x) (mklist x) (flatten x)))
          x))

(defun prune (test tree)
  (if (atom tree)
      tree
      (mapcar #'(lambda (x)
                  (prune test x))
              (remove-if #'(lambda (y)
                             (and (atom y)
                                  (funcall test y)))
                         tree))))
```

49 Written as it is, `find2` will generate an error if it runs off the end of a dotted list:

```
> (find2 #'oddp '(2 . 3))
>>Error: 3 is not a list.
```

CLTL2 (p. 31) says that it is an error to give a dotted list to a function expecting a list. Implementations are not required to detect this error; some do, some don't.

The situation gets murky with functions that take sequences generally. A dotted list is a cons, and conses are sequences, so a strict reading of CLTL would seem to require that

```
(find-if #'oddp '(2 . 3))
```

return `nil` instead of generating an error, because `find-if` is supposed to take a sequence as an argument.

Implementations vary here. Some generate an error anyway, and others return `nil`. However, even implementations which follow the strict reading in the case above tend to deviate in e.g. the case of `(concatenate 'cons '(a . b) '(c . d))`, which is likely to return `(a c . d)` instead of `(a c)`.

In this book, the utilities which expect lists expect proper lists. Those which operate on sequences will accept dotted lists. However, in general it would be asking for trouble to pass dotted lists to any function that wasn't specifically intended for use on them.

66 If we could tell how many parameters each function had, we could write a version of `compose` so that, in $f \circ g$, multiple values returned by g would become the corresponding arguments to f. In CLTL2, the new function `function-lambda-expression` returns a lambda-expression representing the original source code of a function. However, it has the option of returning `nil`, and usually does so for built-in functions. What we really need is a function that would take a function as an argument and return its parameter list.

73 A version of `rfind-if` which searches for whole subtrees could be defined as follows:

```
(defun rfind-if (fn tree)
  (if (funcall fn tree)
      tree
      (if (atom tree)
          nil
          (or (rfind-if fn (car tree))
              (and (cdr tree) (rfind-if fn (cdr tree)))))))
```

The function passed as the first argument would then have to apply to both atoms and lists:

```
> (rfind-if (fint #'atom #'oddp) '(2 (3 4) 5))
3
> (rfind-if (fint #'listp #'cddr) '(a (b c d e)))
(B C D E)
```

95 McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *Lisp 1.5 Programmer's Manual,* 2nd Edition. MIT Press, Cambridge, 1965, pp. 70-71.

106 When Section 8.1 says that a certain kind of operator can only be written as a macro, it means, can only be written by the user as a macro. Special forms can do everything macros can, but there is no way to define new ones.

A special form is so called because its evaluation is treated as a special case. In an interpreter, you could imagine `eval` as a big `cond` expression:

```
(defun eval (expr env)
  (cond ...
        ((eq (car expr) 'quote) (cadr expr))
        ...
        (t (apply (symbol-function (car expr))
                  (mapcar #'(lambda (x)
                              (eval x env))
                          (cdr expr))))))
```

Most expressions are handled by the default clause, which says to get the function referred to in the car, evaluate all the arguments in the cdr, and return the result of applying the former to the latter. However, an expression of the form `(quote x)` should not be treated this way: the whole point of a quote is that its argument is not evaluated. So `eval` has to have one clause which deals specifically with `quote`.

Language designers regard special forms as something like constitutional amendments. It is necessary to have a certain number, but the fewer the better. The special forms in Common Lisp are listed in CLTL2, p. 73.

The preceding sketch of `eval` is inaccurate in that it retrieves the function before evaluating the arguments, whereas in Common Lisp the order of these two operations is deliberately unspecified. For a sketch of `eval` in Scheme, see Abelson and Sussman, p. 299.

115 It's reasonable to say that a utility function is justified when it pays for itself in brevity. Utilities written as macros may have to meet a stricter standard. Reading macro calls can be more difficult than reading function calls, because they can violate the Lisp evaluation rule. In Common Lisp, this rule says that the value of an expression is the result of calling the function named in the car on the arguments given in the cdr, evaluated left-to-right. Since functions all follow this rule, it is no more difficult to understand a call to `find2` than to `find-books` (page 42).

However, macros generally do not preserve the Lisp evaluation rule. (If one did, you could have used a function instead.) In principle, each macro defines its own evaluation rule, and the reader can't know what it is without reading the macro's definition. So a macro, depending on how clear it is, may have to save much more than its own length in order to justify its existence.

126 The definition of `for` given in Figure 9.2, like several others defined in this book, is correct on the assumption that the initforms in a `do` expression will be evaluated left-to-right. CLTL2 (p. 165) says that this holds for the stepforms, but says nothing one way or the other about the initforms.

There is good cause to believe that this is merely an oversight. Usually if the order of some operations is unspecified, CLTL will say so. And there is no reason that the order of evaluation of the initforms of a `do` should be unspecified, since the evaluation of a `let` is left-to-right, and so is the evaluation of the stepforms in `do` itself.

128 Common Lisp's `gentemp` is like `gensym` except that it interns the symbol it creates. Like `gensym`, `gentemp` maintains an internal counter which it uses to make print names. If the symbol it wants to create already exists in the current package, it increments the counter and tries again:

```
> (gentemp)
T1
> (setq t2 1)
1
> (gentemp)
T3
```

and so tries to ensure that the symbol created will be unique. However, it is still possible to imagine name conflicts involving symbols created by `gentemp`. Though `gentemp` can guarantee to produce a symbol not seen before, it cannot foresee what symbols might be encountered in the future. Since `gensym`s work perfectly well and are always safe, why use `gentemp`? Indeed, for macros the only advantage of `gentemp` is that the symbols it makes can be written out and read back in, and in such cases they are certainly not guaranteed to be unique.

131 The capture of function names would be a more serious problem in Scheme, due to its single name-space. Not until 1991 did the Scheme standard suggest any official way of defining macros. Scheme's current provision for hygienic macros differs greatly from `defmacro`. For details, and a bibliography of recent research on the subject, see the most recent Scheme report.

137 Miller, Molly M., and Eric Benson. *Lisp Style and Design*. Digital Press, Bedford (MA), 1990, p. 86.

158 Instead of writing `mvpsetq`, it would be cleaner to define an inversion for `values`. Then instead of

```
(mvpsetq (w x) (values y z) ...)
```

we could say

```
(psetf (values w x) (values y z) ...)
```

Defining an inversion for `values` would also render `multiple-value-setq` unnecessary. Unfortunately, as things stand in Common Lisp it is impossible to define such an inversion; `get-setf-method` won't return more than one store variable, and presumably the expansion function of `psetf` wouldn't know what to do with them if it did.

180 One of the lessons of `setf` is that certain classes of macros can hide truly enormous amounts of computation and yet leave the source code perfectly comprehensible. Eventually `setf` may be just one of a class of macros for programming with assertions.

For example, it might be useful to have a macro `insist` which took certain expressions of the form (*predicate* . *arguments*), and would make them true if they weren't already. As `setf` has to be told how to invert references, this macro would have to be told how to make expressions true. In the general case, such a macro call might amount to a call to Prolog.

198 Gelernter, David H., and Suresh Jagannathan. *Programming Linguistics.* MIT Press, Cambridge, 1990, p. 305.

199 Norvig, Peter. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, San Mateo (CA), 1992, p. 856.

213 The constant `least-negative-normalized-double-float` and its three cousins have the longest names in Common Lisp, with 38 characters each. The operator with the longest name is `get-setf-method-multiple-value`, with 30.

The following expression returns a list, from longest to shortest, of all the symbols visible in the current package:

```
(let ((syms nil))
  (do-symbols (s)
    (push s syms))
  (sort syms
        #'(lambda (x y)
             (> (length (symbol-name x))
                (length (symbol-name y))))))
```

217 As of CLTL2, the expansion function of a macro is supposed to be defined in the environment where the `defmacro` expression appears. This should make it possible to give `propmacro` the cleaner definition:

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
     `(get ,obj ',propname)))
```

But CLTL2 does not explicitly state whether the `propname` form originally passed to `propmacro` is part of the lexical environment in which the inner `defmacro` occurs. In principle, it seems that if `color` were defined with `(propmacro color)`, it should be equivalent to:

```
(let ((propname 'color))
  (defmacro color (obj)
    `(get ,obj ',propname)))
```

or

```
(let ((propname 'color))
  (defmacro color (obj)
    (list 'get obj (list 'quote propname))))
```

However, in at least some CLTL2 implementations, the new version of `propmacro` does not work.

In CLTL1, the expansion function of a macro was considered to be defined in the null lexical environment. So for maximum portability, macro definitions should avoid using the enclosing environment anyway.

238 Functions like `match` are sometimes described as doing unification. They don't, quite; `match` will successfully match `(f ?x)` and `?x`, but those two expressions should not unify.

For a description of unification, see: Nilsson, Nils J. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971, pp. 175-178.

244 It's not really necessary to set unbound variables to gensyms, or to call `gensym?` at runtime. The expansion-generating code in Figures 18.7 and 18.8 could be written to keep track of the variables for which binding code had already been generated. To do this the code would have to be turned inside-out, however: instead of generating the expansion on the way back up the recursion, it would have to be accumulated on the way down.

244 A symbol like `?x` occurring in the pattern of an `if-match` always denotes a new variable, just as a symbol in the car of a `let` binding clause does. So although Lisp variables can be used in patterns, pattern variables from outer queries cannot—you can use the same *symbol,* but it will denote a new variable. To test that two lists have the same first element, it wouldn't work to write:

```
(if-match (?x . ?rest1) lst1
    (if-match (?x . ?rest2) lst2
        ?x))
```

In this case, the second `?x` is a new variable. If both `lst1` and `lst2` had at least one element, this expression would always return the car of `lst2`.

However, since you can use (non-`?`ed) Lisp variables in the pattern of an `if-match`, you can get the desired effect by writing:

```
(if-match (?x . ?rest1) lst1
    (let ((x ?x))
      (if-match (x . ?rest2) lst2
          ?x)))
```

The restriction, and the solution, apply to the `with-answer` and `with-inference` macros defined in Chapters 19 and 24 as well.

254 If it were a problem that "unbound" pattern variables were `nil`, you could have them bound to a distinct gensym by saying `(defconstant unbound (gensym))` and then replacing the line

```
`(,v (binding ',v ,binds)))
```

in `with-answer` with:

```
`(,v (aif2 (binding ',v ,binds) it unbound))
```

258 Scheme was invented by Guy L. Steele Jr. and Gerald J. Sussman in 1975. The language is currently defined by: Clinger, William, and Jonathan A. Rees (Eds.). *Revised[4] Report on the Algorithmic Language Scheme.* 1991.

This report, and various implementations of Scheme, were at the time of printing available by anonymous FTP from `altdorf.ai.mit.edu:pub`.

266 As another example of the technique presented in Chapter 16, here is the derivation of the `defmacro` template within the definition of `=defun`:

```
(defmacro fun (x)
  `(=fun *cont* ,x))

(defmacro fun (x)
  (let ((fn '=fun))
    `(,fn *cont* ,x)))

`(defmacro ,name ,parms
   (let ((fn ',f))
     `(,fn *cont* ,,@parms)))

`(defmacro ,name ,parms
   `(,',f *cont* ,,@parms))
```

267 If you wanted to see multiple return values in the toplevel, you could say instead:

```
(setq *cont*
      #'(lambda (&rest args)
          (if (cdr args) args (car args))))
```

273 This example is based on one given in: Wand, Mitchell. Continuation-Based Program Transformation Strategies. *JACM* 27, 1 (January 1980), pp. 166.

273 A program to transform Scheme code into continuation-passing style appears in: Steele, Guy L., Jr. LAMBDA: The Ultimate *Declarative.* MIT Artificial Intelligence Memo 379, November 1976, pp. 30-38.

292 These implementations of `choose` and `fail` would be clearer in T, a dialect of Scheme which has `push` and `pop`, and allows `define` in non-toplevel contexts:

```
(define *paths* ())
(define failsym '@)

(define (choose choices)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (push *paths*
                (lambda () (cc (choose (cdr choices)))))
          (car choices)))))

(call-with-current-continuation
  (lambda (cc)
    (define (fail)
      (if (null? *paths*)
          (cc failsym)
          ((pop *paths*))))))
```

For more on T, see: Rees, Jonathan A., Norman I. Adams, and James R. Meehan. *The T Manual*, 5th Edition. Yale University Computer Science Department, New Haven, 1988.

The T manual, and T itself, were at the time of printing available by anonymous FTP from `hing.lcs.mit.edu:pub/t3.1`.

293 Floyd, Robert W. Nondeterministic Algorithms. *JACM* 14, 4 (October 1967), pp. 636-644.

298 The continuation-passing macros defined in Chapter 20 depend heavily on the optimization of tail calls. Without it they may not work for large problems. For example, at the time of printing, few computers have enough memory to allow the Prolog defined in Chapter 24 to run the zebra benchmark without the optimization of tail calls. (Warning: some Lisps crash when they run out of stack space.)

303 It's also possible to define a depth-first correct *choose* that works by explicitly avoiding circular paths. Here is a definition in T:

```
(define *paths* ())
(define failsym '@)
(define *choice-pts* (make-symbol-table))

(define-syntax (true-choose choices)
  `(choose-fn ,choices ',(generate-symbol t)))
```

```
(define (choose-fn choices tag)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (push *paths*
                (lambda () (cc (choose-fn (cdr choices)
                                          tag))))
          (if (mem equal? (car choices)
                          (table-entry *choice-pts* tag))
              (fail)
              (car (push (table-entry *choice-pts* tag)
                         (car choices)))))))))
```

In this version, `true-choose` becomes a macro. (The T `define-syntax` is like `defmacro` except that the macro name is put in the car of the parameter list.) This macro expands into a call to `choose-fn`, a function like the depth-first `choose` defined in Figure 22.4, except that it takes an additional `tag` argument to identify choice-points. Each value returned by a `true-choose` is recorded in the global hash-table `*choice-pts*`. If a given `true-choose` is about to return a value it has already returned, it fails instead. There is no need to change `fail` itself; we can use the `fail` defined on page 396.

This implementation assumes that paths are of finite length. For example, it would allow `path` as defined in Figure 22.13 to find a path from `a` to `e` in the graph displayed in Figure 22.11 (though not necessarily a direct one). But the `true-choose` defined above wouldn't work for programs with an infinite search-space:

```
(define (guess x)
  (guess-iter x 0))

(define (guess-iter x g)
  (if (= x g)
      g
      (guess-iter x (+ g (true-choose '(-1 0 1))))))
```

With `true-choose` defined as above, `(guess` *n*`)` would only terminate for non-positive *n*.

How we define a correct *choose* also depends on what we call a choice point. This version treats each (textual) call to `true-choose` as a choice point. That might be too restrictive for some applications. For example, if `two-numbers` (page 291) used this version of *choose*, it would never return the same pair of numbers twice, even if it was called by several different functions. That might or might not be what we want, depending on the application.

Note also that this version is intended for use only in compiled code. In interpreted code, the macro call might be expanded repeatedly, each time generating a new gensymed tag.

305 Woods, William A. Transition Network Grammars for Natural Language Analysis. *CACM* 3, 10 (October 1970), pp. 591-606.

312 The original ATN system included operators for manipulating registers on the stack while in a sub-network. These could easily be added, but there is also a more general solution: to insert a lambda-expression to be applied to the register stack directly into the code of an arc body. For example, if the node `mods` (page 316) had the following line inserted into the body of its outgoing arc,

```
(defnode mods
  (cat n mods/n
    ((lambda (regs)
       (append (butlast regs) (setr a 1 (last regs)))))
    (setr mods *)))
```

then following the arc (however deep) would set the the topmost instance of the register `a` (the one visible when traversing the topmost ATN) to `1`.

323 If necessary, it would be easy to modify the Prolog to take advantage of an existing database of facts. The solution would be to make `prove` (page 336) a nested *choose:*

```
(=defun prove (query binds)
  (choose
     (choose-bind b2 (lookup (car query) (cdr query) binds)
       (=values b2))
     (choose-bind r *rules*
       (=funcall r query binds))))
```

325 To test quickly whether there is any match for a query, you could use the following macro:

```
(defmacro check (expr)
  `(block nil
     (with-inference ,expr
       (return t))))
```

344 The examples in this section are translated from ones given in: Sterling, Leon, and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, 1986.

349 The lack of a distinct name for the concepts underlying Lisp may be a serious barrier to the language's acceptance. Somehow one can say "We need to use C++ because we want to do object-oriented programming," but it doesn't sound nearly as convincing to say "We need to use Lisp because we want to do Lisp programming."

To administrative ears, this sounds like circular reasoning. Such ears would rather hear that Lisp's value hinged on a single, easily understood concept. For years we have tried to oblige them, with little success. Lisp has been described as a "list-processing language," a language for "symbolic computation," and most recently, a "dynamic language." None of these phrases captures more than a fraction of what Lisp is about. When retailed through college textbooks on programming languages, they become positively misleading.

Efforts to sum up Lisp in a single phrase are probably doomed to failure, because the power of Lisp arises from the combination of at least five or six features. Perhaps we should resign ourselves to the fact that the only accurate name for what Lisp offers is *Lisp*.

352 For efficiency, `sort` doesn't guarantee to preserve the order of sequence elements judged equal by the function given as the second argument. For example, a valid Common Lisp implementation could do this:

```
> (let ((v #((2 . a) (3 . b) (1 . c) (1 . d))))
    (sort (copy-seq v) #'< :key #'car))
#((1 . D) (1 . C) (2 . A) (3 . B))
```

Note that the relative order of the first two elements has been reversed.

The built-in `stable-sort` provides a way of sorting which won't reorder equal elements:

```
> (let ((v #((2 . a) (3 . b) (1 . c) (1 . d))))
    (stable-sort (copy-seq v) #'< :key #'car))
#((1 . C) (1 . D) (2 . A) (3 . B))
```

It is a common error to assume that `sort` works like `stable-sort`. Another common error is to assume that `sort` is nondestructive. In fact, both `sort` and `stable-sort` can alter the sequence they are told to sort. If you don't want this to happen, you should sort a copy. The call to `stable-sort` in `get-ancestors` is safe because the list to be sorted has been freshly made.

Index