title 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.