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?*