From Essentials of Programming Languages, 2nd. ed., Friedman et al.

Foreword

This book brings you face-to-face with the most fundamental idea in computer programming:

The interpreter for a computer language is just another program.

It sounds obvious, doesn't it? But the implications are profound. If you are a computational theorist, the interpreter idea recalls Gödel's discovery of the limitations of formal logical systems, Turing's concept of a universal computer, and von Neumann's basic notion of the stored-program machine. If you are a programmer, mastering the idea of an interpreter is a source of great power. It provokes a real shift in mindset, a basic change in the way you think about programming.

I did a lot of programming before I learned about interpreters, and I produced some substantial programs. One of them, for example, was a large data-entry and information-retrieval system written in PL/I. When I implemented my system, I viewed PL/I as a fixed collection of rules established by some unapproachable group of language designers. I saw my job as not to modify these rules, or even to understand them deeply, but rather to pick through the (very) large manual, selecting this or that feature to use. The notion that there was some underlying structure to the way the language was organized, and that I might want to override some of the language designers' decisions, never occurred to me. I didn't know how to create embedded sublanguages to help organize my implementation, so the entire program seemed like a large, complex mosaic, where each piece had to be carefully shaped and fitted into place, rather than a cluster of languages, where the pieces could be flexibly combined. If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master.

There are three reasons why as a programmer you should learn about interpreters.

First, you will need at some point to implement interpreters, perhaps not interpreters for full-blown general-purpose languages, but interpreters just the same. Almost every complex computer system with which people interact in flexible ways—a computer drawing tool or an information-retrieval system, for example—includes some sort of interpreter that structures the interaction. These programs may include complex individual operations—shading a region on the display screen, or performing a database search—but the interpreter is the glue that lets you combine individual operations into useful patterns. Can you use the result of one operation as the input to another operation? Can you name a sequence of operations? Is the name local or global? Can you parameterize a sequence of operations, and give names to its inputs? And so on. No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system. It's easy to find examples of programs with good individual operations, but lousy glue; looking back on it, I can see that my PL/I database program certainly had lousy glue.

Second, even programs that are not themselves interpreters have important interpreter-like pieces. Look inside a sophisticated computer-aided design system and you're likely to find a geometric recognition language, a graphics interpreter, a rule-based control interpreter, and an object-oriented language interpreter all working together. One of the most powerful ways to structure a complex program is as a collection of languages, each of which provides a different perspective, a different way of working with the program elements. Choosing the right kind of language for the right purpose, and understanding the implementation tradeoffs involved: that's what the study of interpreters is about.

The third reason for learning about interpreters is that programming techniques that explicitly involve the structure of language are becoming increasingly important. Today's concern with designing and manipulating class hierarchies in object-oriented systems is only one example of this trend. Perhaps this is an inevitable consequence of the fact that our programs are becoming increasingly complex—thinking more explicitly about languages may be our best tool for dealing with this complexity. Consider again the basic idea: the interpreter itself is just a program. But that program is written in some language, whose interpreter is itself just a program written in some language whose interpreter is itself .... Perhaps the whole distinction between program and programming language is a misleading idea, and future programmers will see themselves not as writing programs in particular, but as creating new languages for each new application.

Friedman, Wand, and Haynes have done a landmark job, and their book will change the landscape of programming-language courses. They don't just *tell* you about interpreters; they *show* them to you. The core of the book is a tour de force sequence of interpreters starting with an abstract high-level language and progressively making linguistic features explicit until we reach a state machine. You can actually run this code, study and modify it, and change the way these interpreters handle scoping, parameter-passing, control structure, etc.

Having used interpreters to study the execution of languages, the authors show how the same ideas can be used to analyze programs without running them. In two new chapters, they show how to implement type checkers and inferencers, and how these features interact in modern object-oriented languages.

Part of the reason for the appeal of this approach is that the authors have chosen a good tool—the Scheme language, which combines the uniform syntax and data-abstraction capabilities of Lisp with the lexical scoping and block structure of Algol. But a powerful tool becomes most powerful in the hands of masters. The sample interpreters in this book are outstanding models. Indeed, since they are *runnable* models, I'm sure that these interpreters and analyzers will find themselves at the cores of many programming systems over the coming years.

This is not an easy book. Mastery of interpreters does not come easily, and for good reason. The language designer is a further level removed from the end user than is the ordinary application programmer. In designing an application program, you think about the specific tasks to be performed, and consider what features to include. But in designing a language, you consider the various applications people might want to implement, and the ways in which they might implement them. Should your language have static or dynamic scope, or a mixture? Should it have inheritance? Should it pass parameters by reference or by value? Should continuations be explicit or implicit? It all depends on how you expect your language to be used, on which kinds of programs should be easy to write, and which you can afford to make more difficult.

Also, interpreters really *are* subtle programs. A simple change to a line of code in an interpreter can make an enormous difference in the behavior of the resulting language. Don't think that you can just skim these programs—very few people in the world can glance at a new interpreter and predict from that how it will behave even on relatively simple programs. So study these programs. Better yet, *run* them—this is working code. Try interpreting some simple expressions, then more complex ones. Add error messages. Modify the interpreters. Design your own variations. Try to really master these programs, not just get a vague feeling for how they work.

If you do this, you will change your view of your programming, and your view of yourself as a programmer. You'll come to see yourself as a designer of languages rather than only a user of languages, as a person who chooses the rules by which languages are put together, rather than only a follower of rules that other people have chosen.

Hal Abelson
Cambridge, MA
August, 2000