Alex Groce (agroce@gmail.com), Northern Arizona University

Brian Kernighan and P. J. Plauger's *The Elements of Programming Style (Second Edition)* starts off with a preface to the second edition, which begins "The practice of computer programming has changed since *The Elements of Programming Style* first appeared." This is indisputably true; the first edition appeared in 1974, and the much-revised second edition, with which we are concerned, appeared in 1978. The only languages in the book are FORTRAN (and often pre-FORTRAN-77-FORTRAN, at that) and PL/I. Has the practice of programming changed so much that this book no longer holds any value for software engineers? Moreover, is it remotely conceivable that a book full of code snippets in outmoded, arcane, all-caps languages could be anything but a good way to turn general readers into fiercely dedicated advocates of never, ever, reading a *Passages* classic again?

To make the argument for this classic, let's begin at the end, which is often a smart way to go.

> "It is not a list of rules so much as an approach and an attitude. 'Good programmers' are those who already have learned a set of rule that ensures good style; many of them will read this book and see no reason to change. If you are still learning to be a 'good programmer,' however, then perhaps some of what we consider good style will have rubbed off in the reading."

The quote is from the epilogue; in fact, these are the final words of the book. The essence of the book is not a list of formal rules for navigating the arcane constraints of PL/I or (heaven help us) old-school FORTRAN. The essence is an approach to programming style informed by the goal of making programs readable *by humans, not just by machines*, and making programs *maintainable over time because they are readable*. While the scale of software systems has grown by leaps and bounds since the book appeared, that scale still includes the expression, control structure, and program/module structure elements that Kernighan and Plaguer addressed. Their approach is to mix pithy short summary rules with example code used to illustrate each concept and elaborate it. The approach, at the time, did not depend on deep working knowledge of PL/I and FORTRAN, and it still doesn't; in fact, I would argue that the lack of familiarity of these languages is helpful, because it forces the modern reader to abstract to the general principles (without losing the advantages of the concrete embodiment of an instance of each principle): it is, thank heavens, not possible to accidentally read this book only as a set of Iron Rules for Writing Java (or C++, or Python).

The first abstract rule given, in the first chapter, is perhaps the most important, and the one still least followed by too many programmers: *"Write clearly – don't be too clever."* This is the heart of good programming style, but what, in practice, does it mean? Perhaps this is too abstract to make the importance of programming style clear to us.

The second chapter, on expressions, serves as a better way to understand the entire book, and see what is meant by rules as concretizing an "approach and an attitude." Let's look closely at

what Kernighan and Plauger have to say about expressions, the atomic building blocks of all our software systems. First, there are the rules:

1. *Say what you mean, simple and directly.*
2. *Use library functions.*
3. *Avoid temporary variables.*
4. *Write clearly – don't sacrifice clarity for "efficiency."*
5. *Let the machine do the dirty work.*
6. *Replace repetitive expressions by calls to a common function.*
7. *Parenthesize to avoid ambiguity.*
8. *Choose variable names that won't be confused.*
9. *[Avoid the FORTRAN arithmetic IF.]*
10. *Avoid unnecessary branches.*
11. *Use the good features of the language; avoid the bad ones.*
12. *Don't use conditional branches as a substitute for a logical expression.*
13. *Use the "telephone test" for readability.*

I have placed in brackets the one rule a modern reader can fairly consider outdated. There are arguably "equivalents" for "modern" languages (do you consider C a modern language? I sometimes fear it will always be a modern language), but the FORTRAN computed GOTO is fortunately long gone, and good riddance to it. Even so, the essence of the rule is captured about as well by rule 11, which is an extremely relevant rule in the era of C++, and if we omit rule 9 we cut our list to a nice, less-scary, 12 rules and avoid bad luck or at least avoid displeasing the triskaidekaphobic.

The rest of the rules are just as valid now, and just as well expressed by the accompanying code, as they were when the book was written. Clarity of individual expressions is still essential to code readability and maintenance, and still all-too-often neglected. In other words, these are really good rules, and a really good approach and attitude. Code is for humans to read before it is for machines to execute. Moreover, hard to read code is more likely to execute wrong, and nobody ever to know until it is too late. And then: what a pain to debug, that code full of obscure, too-clever constructs, poorly and similarly named temp variables, bogus branches, and handcrafted "efficient" substitutes for serviceable library routines!

Speaking of bugs and debugging, this chapter also includes, not by happenstance, what I consider one of the greatest quotations in software engineering, or perhaps the greatest quotation in software engineering: "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" In fact, perhaps this review can stop here. Once a software engineer worth her salt knows this book is where that quote came from, she will likely just run off to Amazon or a used bookstore to find a copy, and devil take the hindmost.

But what of the non-programmer? Can your English professor friend bear to read FORTRAN? I

say yes. The rules, and the text explaining them, actually convey the “language” aspect of software engineering in a way that will appeal to, and educate, anyone who can enjoy Strunk and White. Code is a language; yes, a formal language; but a sonnet is also a formal machine, and oft hard to read (and certainly to make) without training. The details here are safely difficult for all readers, trained and untrained. The concepts illuminate what a computer program really is, and I expect that to remain true for another twenty years or more.