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

Donald E. Knuth's *Literate Programming* is only the second Knuth book I have reviewed here, but his sensibility and example have informed many *Passages* columns, including those on Brooks' *The Mythical Man-Month* and Martin Gardner's recreational mathematics columns.

Literate Programming, in the title, explains why that is so. The central idea of this column series is that software engineering is, to a large extent, a literary endeavor: that we write programs, yes, for computers to execute, but also for human beings to read and understand. The first reason we write such programs specifically for human beings is common between conventional literature and computer programs as literature: the books, poems, short stories, and, yes, computer systems, of the future are composed by reading and understanding and reworking the books, poems, short stories, and, yes, computer systems of the past. There are no illiterate authors; books are made, famously, of other books. Countess Lovelace and Alan Turing, by necessity, wrote programs to a large extent *de novo*, but their actual code was not the most important aspect of their work; such invention is not the common practice of the field.

The recent success of large language models makes this dependence unusually vivid. Although in an important sense they may have little genuine understanding of the problems they solve, they have, in a sense, read an enormous fraction of the world's programs, and that lets them produce useful code in novel settings. They resemble a gifted literary parodist: perhaps unable to invent an entirely original story, but capable of producing a remarkably convincing fusion of *The Odyssey* and P. G. Wodehouse, or aping Hemingway or Chandler very competently indeed. Human authors, of course, do much more than this, but Shakespeare, Joyce, and Eliot all remind us how profoundly originality can depend on deep reading and transformation of base material.

The other reason computer programs must be written for human beings to read is unique to computer programs: all software that survives is modified. In fact, successful software is almost defined by the fact that it spends more time undergoing change than being initially created. This means that human beings, often ones other than the original programmer (or the original programmer, after many months, years, or even decades, and so no longer the same person), must be able to understand it well enough to change it, extend it, and fix it.

I think this book is primarily valuable because it contains Knuth’s best work elaborating this claim, which I think is an important counterpoint to the pragmatic view that computer programs are costly things that control even more costly things (the mainframes and microcomputers of early days, the distributed systems of our day), or the academic view that computer programs are just embodied mathematics. The latter view, which can be (somewhat unfairly) ascribed to Dijkstra and other supporters of formal proof and structured development, is valuable and has an element of lasting intellectual disciplinary power, but can miss out on something startling but true. Namely, that a slightly incorrect, oddly structured, computer program that is easy to understand and think about can be better than a correct, beautifully structured, program that is almost impossible to hold in one’s head. Knuth offers a valuable correction to

overly-mathematical conceptions of the virtues of a good program. That this correction comes from Knuth, who cannot be accused of lacking in rigor or mathematical insight, makes it more powerful and less easy to dismiss. Dijkstra also regarded programs as objects to be understood by human beings, but he emphasized mathematical structure where Knuth emphasized exposition; the mathematical view has weaknesses perhaps more obvious now that a much smaller fraction of the code humans (or LLMs) write is concerned with mathematical operations.

Let me admit that literate programming, itself, was probably a dead end. No truly important programs not written by Don Knuth or part of his TeX or METAFONT family of programs were ever produced in the actual literate style as Knuth envisaged it, and while Knuth's WEB system has github-hosted variants, one with almost 1,000 stars and some contributions within the last five years, I think it's safe to call the specific thing he calls "literate programming" a niche, hobbyist approach with relatively little impact on any current programming language, core tool set, or methodology, though it would be possible to argue that Jupyter notebooks, heavy use of Java/Python docstrings, or even just R Markdown itself have some limited inheritance from Knuth's proposal.

This does not, in my opinion matter, or make this a less important book. First, that WEB's idea of tangling and weaving the same set of files to produce both an essay about a program and a compilable source file did not end up being very important does not mean the ideas motivating WEB didn't matter. Knuth had a big idea: programs should be essays for human readers about how a certain computation can be performed (including notes on decisions and alternatives). That was at least partly right; he ended up presenting that idea in the form of technology that was too tied to the programming tools of his day, and their limitations. The big idea matters, not the particular form of the tools Knuth made.

Second, while the title of the book is *Literate Programming*, the content is more general than that would suggest. This is a book about "Literate Programming," the Knuth proposal that was implemented in the WEB system, but it's also a book about "Literate" "Programming", programming that takes the idea of readability of computer programs seriously, in whatever form. Knuth's defense of *goto* against Dijkstra, his great Turing lecture on "Computer Programming as an Art," and his nearly 100 page exposition of errors made over the years in TeX (essentially what amounts to a curated, brilliant, github change-log for one of the most important computer programs for scientific and mathematical research of all time), are all contained here, and probably make up somewhere between over a third and almost half (depending on what you count) of the content. None of this depends on the narrow WEB proposal at all, but all the contents of the book share a common concern: how programmers actually understand programs, in some cases over decades of use and revision.

I think as a book, *Literate Programming* is valuable to software engineers now because it is the only Knuth work that is mostly focused on issues that belong squarely in the realm of software engineering; while Knuth touches on SE topics in many of his other books, especially his selected papers on computer languages, it is only here that he is mostly engaged in work that can be recognized as software engineering, not touching software engineering as a side effect

while thinking about analysis of algorithms or programming language design. Even the argument with Dijkstra about *goto* is about looping constructs as program organization tools in a way that is more aligned with software engineering concerns than with programming language concerns. Part of Knuth's point is to defend the humble state machine as a tool for organizing human thought about programs, and similar ideas like David Harel's Statecharts really belong more to software engineering than to PL.

Donald E. Knuth would be a giant of computer science even if it were true that he is a mediocre or even bad writer, because he is inarguably a great mathematician and a great designer and developer of useful systems that are still in use. His eminence is elevated to pre-eminence because he is, in fact, a very good writer, and because he has thought long and hard about human problems of software that lie outside his core concerns. This is one of his best books, and probably a more useful read for most software engineers than *The Art of Computer Programming*, however heretical that claim may sound.