The Software Arts

Reviewed by Ragnhild Solberg

The words “computing” and “software” are sure to create some images in your mind. These images might be of machines or data chips, circuit boards or Boolean algebra, perhaps even sentient machines of the science fiction type. In The Software Arts (2019), Warren Sack argues that they should also be of grammar, logic, and rhetoric; in short, the trivium of the liberal arts.

As a contextual frame, the book is part of the ongoing series "Software studies", edited by Matthew Fuller, Lev Manovich, and Noah Wardrip-Fruin. Readers of this review might recognize titles such as 10 Print (with the catchy full title of 10 PRINT CHR$(205.5+RND(1));: GOTO 10, Montfort et al, 2012) which takes a Commodore 64 code as the basis for discussing code as a cultural object, or Programmed Visions (Chun, 2011) that presents how software is intertwined with governmentality. The Software Arts very much speaks to these other entries in the series. A central question driving the book is: What if the history of computing is not what we think? To explore this, the author envisions himself as the narrator of a story where historical and present computers are language machines instead of numerical machines, or, more precisely, “machines of rhetoric, grammar, logic, and dialectic” (18). This story must be told because, argues Sack, software in contemporary cultural and scholarly discourse is framed as a technical entity, far removed from the liberal arts. Thus, the book’s mission is to show how computing grew from the arts, and that the arts are at the center of computing. In Sack’s words, we “need to overcome entrenched divisions in knowledge itself, dividing ‘humanistic’ from ‘technical’ or ‘scientific’ culture” (xiv), and The Software Arts is part of that bridging.

A fundamental assumption is that software is essentially a rewriting or a translation. The humanities understanding that Sack builds on is translation as enabling the exchange of ideas with loss, change, or gain of meaning. Translation thus becomes both the object of study (the software texts of codes and the historic essays written about computers) and the method of analysis (using translation as a way of thinking about software). In order to accomplish this, Sack draws on actor-network theory (ANT), amended with more emphasis on semiotics. His justification is that ANT ethnographies for software and computer history are knowledgeable on programmers but light on semiotics and the texts of software. By looking for contradictions and instabilities in the texts themselves and placing these in their historical context, the author seeks to find what is lost in the act of translation.

The book is composed of eight chapters. Beyond the introduction and conclusion, the chapters are Translation, Language, Algorithm as well as the trivium of Logic, Rhetoric, and Grammar. I would note that the totality would benefit from being read in a physical format. My old Kindle, albeit perfect for reading [fantasy literature, has some issues jumping back and forth between the text and the table of contents. As a result, I spent the majority of the introductory chapter wondering where it all was going, because the text itself wants to do everything at once. It does, however, eventually do almost everything. The author writes that “simply put, this book is a close reading of key texts of computer science and its history” (25), but there is little simple about it. Sack’s generosity in explaining mechanical and liberal arts terms, presenting a comprehensive history of computer texts and their academic environment, and discussing numerous theorists of epistemology interspersed with lines of code and syntactic maps should show how this is a project that reaches beyond its 400 paged binder. Phrased otherwise, it becomes hard to follow at times, which is somewhat strange for a book with rhetoric and language as chapter headings and an intended demography including non-academics. As such, it is certainly a book for those who want a comprehensive dive into pre-digital software history, software as liberal arts, or the relationship between syntax and semiotics. For a broader audience, it is the general ideas presented that are of interest.

The core of the book’s contribution is its rich history. The historical approach to the texts of software through the lens of logic, rhetoric, and grammar is an interesting read that allows the author as a narrator of stories to shine. One such story is how Alan Turing’s “universal machine” is popularized beyond its original meaning (chapter 2). Misreading and popularization is also a form of translation, writes Sack. Going back to the original texts and their historical context, Sack shows that Turing and his contemporary Alonzo Church’s claims are not that all machines can do anything, but about their specific machines working within specific limits. In contrast to many scholarly and popular conceptualizations of Turing machines, writes Sack, Turing’s article shows that there are things these machines cannot do. Lost in translation from this text is the historic understanding of computers as something human, i.e. including the human worker operating and interacting with the machine and other people. Subheadings in The Software Arts such as “when computers were human” followed by “when computers became machines” emphasize this translation shift.
An important source for Sack is the eighteenth century French
*Encyclopédie*, where he finds “the root for programming languages”
(60) in its pairing of mechanical and liberal arts. One story follows
the line from the *Encyclopédie* to how logic was displaced from
the trivium due to, among others, the translation of logic into
arithmetic (chapter 5). This “arithmetization”, which Sack argues
is an urge to make everything into math (that must be resisted),
supposes a universal logic. However, according to Sack, logic is a
language that has undergone several translations. Thus, there are
several logics, one of which is software.

Ultimately, what Sack emphasizes is that all of these stories have
present-day ramifications. He explains that

epistemological divisions led to divisions in the educational
system, where the liberal arts were taught separately from the
mechanical arts. To this day, the Aristotelian barrier separates
language that belongs to the liberal arts (specifically the
language arts of the trivium) from machines that belong to the
mechanical arts. (60)

In this lies not-so-modest implications for education redesign. First,
accepting the book’s premise requires bridging the mechanical
and liberal arts, with the structural and institutional as well as
philosophical changes that will bring. Sack himself suggests,
“software studies should be actively finding ways to go beyond
computer science, to fix computer science’s omissions and
mistakes, and to construct its own research agenda. Interaction,
assignment, equivalence, and identity could be at or near the top
of that agenda.” (258). Second, all texts are and should be read as
translations. The author’s history of software is also a translation,
one in which he is explicit about its role as such. Despite not
acknowledging the rabbit hole of epistemology when software
and logic and basically everything else is translation, in Sack’s
use translation seems to denote the interference and influence
of other agents in what presents as real, of which we all can use
an occasional reminder. For instance, he points interferences in
algorithms constructed with the power to determine equivalences.
While discussing how these algorithms can persuade us (chapter
6), *The Software Arts* nods to (but does not pursue) research that
also bridges liberal and mechanical arts to uncover biases and black
boxes in computational media, such as the work of Virginia Eubanks
(2018) and others. The book’s historical approach will result in older
sources, but making the nod to emerging research in the digital
humanities into a handshake would surely strengthen its argument.

According to Sack, gaps between the narrative of the computer
and its rhetoric equations should force us, like the London tube,
to “mind the gap” (31, 35). Through the gaps, Sack finds several
historical connections between computer science and the liberal
arts. In a sense, *The Software Arts* read like a defense of why we
need software studies. It does so rather convincingly, through
its insistence on debunking popularized conceptualizations of
computation by reading the source material as translations. It
would be interesting to hear what someone from mathematics or
computer science have to say about this translation and whether
it is as convincing to them (even if this proposition might reinforce
the trenches of knowledge that Sack wants to remove). Overall,
what the book does is show that there is value in strengthening the
artistic bonds of how-to-knowledge with software. It reminds us
that words have value; they matter, and they matter in a context.
Through its focus on computers as machines of language and
meaning, *The Software Arts* is an insightful narrative of software’s
integration in society, of the status quo of computational science,
and of what the story could look like if we try to think of and with
software as translation.

**References**

Cambridge, MA: MIT Press.

Eubanks, V. (2018). *Automating Inequality: How High-Tech Tools Profile,

Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglass, J., Marino, Mark
PRINT CHR$(205.5+RND(1));: GOTO 10,* Cambridge, MA: MIT Press.


**Author:** Ragnhild Solberg, PhD candidate
Department of Linguistic, Literary, and Aesthetic Studies, University of Bergen

**Licensing:** All content in NJSTS is published under a Creative Commons Attribution 4.0 license. This means that anyone is free to share (copy and redistribute the material in any medium or format) or adapt (remix, transform, and build upon the material) the material as they like, provided they give appropriate credit, provide a link to the license, and indicate if changes were made.