software studies\ a lexicon

edited by **matthew fuller**

SOFTWARE STUDIES

For information about special quantity discounts, please email special_sales@mitpress.mit.edu

This book was set in Garamond 3 and Bell Gothic by Graphic Composition, Inc.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data
Software studies : a lexicon / edited by Matthew Fuller.
p. cm.—(Leonardo books)
Includes bibliographical references and index.
ISBN 978-0-262-06274-9 (hbk. : alk. paper) 1. Computer software. 2. Computers and civilization—Encyclopedias. 3. Programming languages (Electronic computers)—Lexicography. 4. Technology and the arts. I. Fuller, Matthew.
QA76.754.S64723 2008
005.1—dc22

2007039724

10 9 8 7 6 5 4 3 2 1

Algorithm

Andrew Goffey

Algorithm = Logic + Control[1]

The importance of the algorithm for software studies is indicated with admirable succinctness by Les Goldschlager and Andrew Lister in their textbook, *Computer Science: A Modern Introduction*. The algorithm "is the unifying concept for all the activities which computer scientists engage in." Provisionally a "description of the method by which a task is to be accomplished," the algorithm is thus the fundamental entity with which computer scientists operate.[2] It is independent of programming languages and independent of the machines that execute the programs composed from these algorithms. An algorithm is an abstraction, having an autonomous existence independent of what computer scientists like to refer to as "implementation details," that is, its embodiment in a particular programming language for a particular machine architecture (which particularities are thus considered irrelevant).

But the algorithm is not simply the theoretical entity studied by computer scientists. Algorithms have a real existence embodied in the class libraries of programming languages, in the software used to render web pages in a browser (indeed, in the code used to render a browser itself on a screen), in the sorting of entries in a spreadsheet and so on. Specialized fields of research, such as artificial life or connectionism in cognitive science, utilize genetic algorithms, backpropagation algorithms, least mean square algorithms for the construction of models to simulate evolutionary processes or the learning capacities of neural networks. Algorithms have material effects on end users—and not just when a commercial website uses data-mining techniques to predict your shopping preferences.

In short, both theoretically and practically, ideally and materially, algorithms have a crucial role in software. But none of this tells us much about the social, cultural, and political role algorithms play, if anything. Nor does it tell us much about the strata of material reality algorithmic abstractions might be correlated with: glowing configurations of pixels on a screen? mouse movements? the flow of electrons around an integrated circuit? Locating itself squarely on the side of the reductionist strategies of the exact sciences, society, culture, and politics are very much marginal to the concerns of computing

science. Software engineering, on the other hand, concerned as it is with the pragmatic efficacy of building software for particular purposes, might appear to offer a better starting point for factoring culture back into software. However, it is unlikely that software engineering will allow us to view culture as anything other than something that software plugs into, as long as we fail to arrive at a better understanding of some of its basic building blocks. The key question then is what, if anything, a study of algorithms as such can tell us about the place of culture in software.

Historically, the algorithm occupies the central position in computing science because of the way that it encapsulates the basic logic behind the Turing machine. Alan Turing's concept of a machine that could be used to determine whether any particular problem is susceptible to being solved mechanically was a highly original interpretation of the aim of David Hilbert's famous project of formally deciding whether or not any mathematical proposition can be proved true. The algorithm, which Turing understood as an effective process for solving a problem, is merely the set of instructions fed into the machine to solve that problem.[3] Without the algorithm then, there would be no computing.

Although computer scientists work with them as if they were purely formal beings of reason (with a little bit of basic mathematical notation, it is possible to reason about algorithms, their properties and so on, the way one can reason about other mathematical entities), algorithms bear a crucial, if problematic, relationship to material reality. This was tacit in the way that the Turing machine was envisaged in terms of effective processes: A computer is a machine, after all, and while the Turing machine is an imaginative abstraction, its connotations of materiality are entirely real. Robert Rosen has suggested that the temptation to extrapolate from formal procedures to material processes was practically inherent in the enterprise of the early computing scientists.[4] Such a temptation implies a confusion between the mathematics of algorithms and the physics of real processes, of which Stephen Wolfram's bold speculation that the universe is itself a giant computer is one possible outcome.[5] The rest of this article explores another possibility, equally speculative but perhaps more mundane.

One of the implications of characterizing the algorithm as a sum of logic and control is that it is suggestive of a link between algorithms and action. Despite the formal-logical framework of the theory of algorithms and the fact that programming languages are syntactic artifacts, the construction of algorithms as a precisely controlled series of steps in the accomplishment of a task is a clear indication of what might be called the pragmatic dimension of

programming. Algorithms do things, and their syntax embodies a command structure to enable this to happen. After all, the Turing machine as an imaginative abstraction had as a material correlate a series of real computers. And dumb though they may be, missile guidance systems, intelligence databases, and biometric testing are all perfectly real. Without this effective existence in concrete machinery, algorithms would only ever have a paper reality as the artifacts of a formal language.

In the field of linguistics, the existence of a pragmatic dimension to language—the fact that words do things—has created enormous problems for attempts to formalize the structure of natural language. Because pragmatics connects language to extrinsic factors, it becomes impossible to conceptualize a language as a self-sufficient system closed in on itself. Perhaps attempting to conceptualize the pragmatic dimension of the algorithm might yield a similar result? However, while formalization comes afterwards with natural languages, with algorithms, formalization comes first, the express aim being to divorce (formal) expression from (material) content completely. Understandably then, the study of computation has tended to concentrate on issues of syntax and semantics, the assumption being that what algorithms do can be appropriately grasped within such a framework. This has tended to result in making the leap from the theoretical world to the practical world a difficult one to accomplish. Always the trivia of implementation details.

A conception of the algorithm as a *statement* as Michel Foucault used the term might allow us to understand this approach a little better. For Foucault, the statement is not analytically reducible to the syntactic or semantic features of a language; it refers instead to its historical existence and the way that this historical existence accomplishes particular actions. The statement is a sort of diagonal line tracing out a function of the existence of language, which is in excess of its syntactic and semantic properties. In this way, the concept of the statement acts as a reminder that the categorical distinction between form and content is, paradoxically, insufficiently abstract to grasp the intelligence of concretely singular constellations of language in their effective existence. As Foucault puts it in *The Archaeology of Knowledge,* "to speak is to do something—something other than to express what one thinks, to translate what one knows, and something other than to play with the structure of language."[6] For Foucault, these actions are restricted to the human sphere, as is only to be expected from an analysis which focuses on the historical existence of natural languages. Appropriately translated into the field of software studies, however,

focusing on the development and deployment of algorithms and an analysis of the actions they accomplish both within software and externally might lead us to view the latter as a sort of machinic discourse, which addresses the ways in which algorithms operate transversally, on themselves, on machines, and on humans. (Alternatively, we might want to start to think about cultural analysis as a process of software engineering.)

Viewing algorithms in this way as statements within a machinic discourse would problematize their existence in a way which undercuts the "pure/applied" or "theory/practice" dichotomies which crop up when the distinction between computing science and software engineering is too hastily made. The formalist aim at complete abstraction from content not only relays the theory/practice divide, it also tends to preclude an analysis of the link between the crucial entities of computing science and historical context. Just because the development of an algorithm requires a level of de facto formal abstraction, which then allows that algorithm to be applied to other kinds of content, does not mean that we have exhausted everything that we need to know to understand the processes of which it is a part. To borrow an expression from Gilles Deleuze and Félix Guattari, whose analysis of the place of pragmatics in language is part of the inspiration for this discussion, the problem with the purely formal conception of the algorithm as an abstract machine is not that it is abstract. It is that it is not abstract enough. That is to say, it is not capable of understanding the place of the algorithm in a process which traverses machine and human.[7]

Algorithms obviously do not execute their actions in a void. It is difficult to understand the way they work without the simultaneous existence of data structures, which is also to say data. Even the simplest algorithm for sorting a list of numbers supposes an unsorted list as input and a sorted list as output (assuming the algorithm is correct). Although computer scientists reason about algorithms independendently of data structures, the one is pretty near useless without the other. In other words, the distinction between the two is formal. However, from a practical point of view, the prerequisite that structured data actually exist in order for algorithms to be operable is quite fundamental, because it is indicative of a critical operation of translation that is required for a problem to be tractable within software. That operation of translation might be better understood as an incorporeal transformation, a transformation that, by recoding things, actions, or processes as information, fundamentally changes their status. This operation can be accomplished in myriad ways, but generally requires a structuring of data, whether by something as innocuous as the use of

a form on a web page or by social processes of a more complex form: the knowledge extraction practiced by the developers of expert systems, the restructuring of an organization by management consultants, and so on.

It would be easy to leave the analysis of algorithms at this point: We are back on familiar territory for cultural analysis, that of the critique of abstraction. Within cultural studies and many other fields of research in the human sciences, abstraction is often thought of as the enemy. Many movements of philosophical thought, literary and artistic endeavor, and human-scientific research set themselves up against the perceived dehumanizing and destructive consequences of the reductionism of mathematics, physics, and allied disciplines, as the perennial debates about the differences between the human and the exact sciences suggests. We could even understand major elements of the concept of culture as a response to the abstract machinery of industrial capitalism and the bifurcated nature modern rationality is built upon. Understanding things, activities, tasks, and events in algorithmic terms appears only to exacerbate this situation. What is an algorithm if not the conceptual embodiment of instrumental rationality within real machines?

However, to simply negate abstraction by an appeal to some other value supposedly able to mitigate the dehumanizing consequences of reductionism misses a crucial point. It fails to adequately question the terms by which the algorithm, as a putatively self-sufficient theoretical construct, maintains its hierarchizing power. In questioning the self-sufficiency of the algorithm as a formal notion by drawing attention to its pragmatic functioning, however, it becomes possible to consider the way that algorithms work as part of a broader set of processes. Algorithms act, but they do so as part of an ill-defined network of actions upon actions, part of a complex of power-knowledge relations, in which unintended consequences, like the side effects of a program's behavior, can become critically important.[8] Certainly the formal quality of the algorithm as a logically consistent construction bears with it an enormous power—particularly in a techno-scientific universe—but there is sufficient equivocation about the purely formal nature of this construct to allow us to understand that there is more to the algorithm than logically consistent form.

Lessig has suggested that "code is law," but if code is law it is law as a "management of infractions."[9] Formal logics are inherently incomplete and indiscernibles exist. Machines break down, programs are buggy, projects are abandoned and systems hacked. And, as the philosopher Alfred North Whitehead has shown, humans are literally infected by abstractions.[10] This no bad

thing, because like the virus which produced variegated tulips of a rare beauty, infection can be creative too.

Notes

1. Robert Kowalski, "Algorithm = logic + control."

2. Les Goldschlager and Andrew Lister, *Computer Science: A Modern Introduction*, 2nd ed., 12.

3. See Rolf Herken, ed., *The Universal Turing Machine: A Half-Century Survey* for an excellent collection of appraisals of the Turing machine.

4. Robert Rosen, "Effective Processes and Natural Law" in Herken, ibid.

5. Stephen Wolfram, *A New Kind of Science*.

6. Replace the word "speak" with the word "program" and one might begin to get a sense of what is being suggested here. See Michel Foucault, *The Archaeology of Knowledge*.

7. Gilles Deleuze and Félix Guattari, "November 20, 1923: Postulates of Linguistics," in *A Thousand Plateaus*.

8. See Philip Agre, *Computation and Human Experience* on the crucial role of side effects in software. Max Weber's essay *The Protestant Ethic and the Spirit of Capitalism* is the classic text on the fundamental role of unintended consequences in human action.

9. Gilles Deleuze, *Foucault*, p. 39.

10. See for example, Alfred North Whitehead, *Science and the Modern World*, and the extended commentary by Isabelle Stengers, *Penser avec Whitehead*.