Methodology of Computer Science

Timothy R. Colburn

To appear as chapter 24 in
The Blackwell Guide to the Philosophy of Computing and Information

Introduction

Science and philosophy are often distinguished by pointing out that science seeks explanation while philosophy seeks justification. To ask what accounts for the neuronal firing of synapses in the brain, for example, is a scientific question, while to ask what would constitute adequate grounds for believing that an artificially constructed neural network is conscious is a philosophical one. So philosophy has been characterized as the critical evaluation of beliefs through the analysis of concepts in a given area of inquiry. Of course, science is also concerned with critically evaluating beliefs and analyzing concepts. However, philosophy is a non-empirical, or a priori, discipline, in distinct contrast with science.

Computer science would seem to be distinguished from philosophy just as any other science. But computer science is unique among the sciences in the types of models it creates. In seeking explanations, science often constructs models to test hypotheses for explaining phenomena. These models, in the form of experimental apparatus, are of course physical objects. The models built and manipulated in computer science, however, are not physical at all. Computer science is a science concerned with the study of computational processes. A computational process is distinguished from, say, a chemical or electrical process, in that it is studied "in ways that ignore its physical nature." (Hailperin et al, 1999, p. 3) For example, the process by which a card player arranges cards in her hand, and the process by which a computer sorts names in a customer list, though they share nothing in common physically, may nevertheless embody the same computational process. They may, for example, both proceed by scanning the items to be arranged one by one, determining the proper place of each scanned item relative to the items already scanned, and inserting it into that place, perhaps necessitating the moving of previously scanned items to make room. This process (known as an insertion sort in computer science terms) can be precisely described in a formal language without talking about playing cards or semiconducting elements. When so described, one has a computational model of the process in the form of a computer program. This model can be tested, in a way analogous to how a hypothesis is tested in the natural sciences, by executing the program and observing its behavior. It can also be reasoned about abstractly, so that questions can be answered about it, such as, whether there are other processes which will have the same effect but achieve it more efficiently. Building computational models and answering these kinds of questions form a large part of what computer scientists do.

The explosive growth in the number of computer applications in the last several decades has shown that there is no limit to how many real-world processes are amenable to modeling by computer. Not only have traditional activities, like record keeping, investing,

publishing, and banking, been simply converted to control by computational models, but whole new kinds of activity have been created that would not be possible without such models. These are the by-now-familiar "virtual" activities described in the language of cyberspace: e-mail, chat rooms, Web surfing, on-line shopping, Internet gaming, and so on.

The role of philosophy in that sub-field of computer science known as artificial intelligence (AI) has long been recognized, given the roles of knowledge and reasoning in AI. And the reverse, the role of AI in philosophy, has even been highlighted by some, albeit controversially. (See the chapters on mind and AI by Fetzer and McLaughlin in this volume.) But apart from considerations arising from the modeling of knowledge and reason, computer science is ripe for the good old-fashioned analysis that philosophy can provide for any science. Thus, a perfectly reasonable role of philosophy is to attempt to place computer science within the broad spectrum of inquiry that constitutes science. The concern here is to deal with the inevitable identity crises that crop up in the self-image of any adolescent, which computer science certainly is. Philosophy should address questions like: What is the relation between mathematics and computer science? Is there a sense in which computer science is experimental science? Is a computer programmer merely a data wizard, or can she also engage in information modeling? What is the nature of abstraction in computer science? What are the ontological implications of computer science concepts? From the point of view of computer science methodology, the most probing of these questions concerns the relation between mathematics and computer science and the nature of abstraction in computer science. The remainder of this chapter turns its attention to these issues.

Computer Science and Mathematics

Philosophical contributions to the foundations of scientific disciplines often center around "pivotal questions" regarding reductionist attempts. In the philosophy of biology, for example, the question is whether the science of the organic can be reduced to the science of the inorganic (the reduction of biological to physical laws). In mathematics, logicism claims that all of mathematics can be reduced to logic. For science in general, logical positivism advocated the reduction of theoretical vocabulary to observational vocabulary. An early "pivotal question" in the philosophy of computer science is whether computer science be reduced to a branch of mathematics. How a computer scientist answers this question can greatly influence his or her methodology.

The range of perspectives from which the reductionist issue can be addressed is wide. Consider the following view, expressed by C.A.R. Hoare: "Computer programs are mathematical expressions. They describe, with unprecedented precision and in the most minute detail, the behavior, intended or unintended, of the computer on which they are executed." (Hoare, 1986, p. 115) And this alternative, offered by C. Floyd: "Programs are tools or working environments for people. [They] are designed in processes of learning and communication so as to fit human needs." (Floyd, 1987, p. 196) The view expressed by Hoare is unequivocal: computer programs are mathematical expressions. The quote by Floyd is less precise, but expresses a view on the function of programs for humans in decidedly non-

mathematical terms. While these views do not necessarily contradict one another, they can most definitely signal contrasting interpretations as to how computer programs ought to be designed, built, and used.

While these quotes express views on the function and status of computer programs, the differences of opinion extend to the broader notion of computing as a science, in which the task of actually creating a program is but one aspect. Of related concern, for example, are the activities of program specification and verification. A program specification is a detailed description of a program's input and output, ignoring the details of how the program actually accomplishes its task. Program verification is the process of determining whether a program actually conforms to its specification. These activities are just as germane to the software process as writing the program itself, and there is no agreement on whether or not program specifications should be mathematical entities and whether or not program verification can be a purely mathematical activity.

There is agreement, however, on the possibility of mathematically reasoning about programs as the *abstract* representations of algorithms, as opposed to programs as the causal manipulations of bits. For example, given a program P consisting of the statements $S_1, \ldots, S_n$, it is possible to construct statements like "Let S_1, S_2, ..., and S_n be an abstract representation of program P. Then P has property R," where R describes some aspect of the execution of P in the abstract sense. For example, R might describe limits on the time it would take P to run, or the amount of memory P would require to execute. By giving precise interpretations to the S_i in a pertinent language and appropriately choosing R, it may be possible that the statement above is a theorem in a formal language. This is in fact the approach taken by modern researchers in computer science who are concerned with reasoning about algorithms and data structures.

While this is a justification of how reasoning *about* programs can be regarded as mathematical, it is yet a much broader claim to say that computer science is, or ought to aspire to be, a branch of mathematics. For there are still the issues of whether the specification, generation, or maintenance of programs (apart from reasoning about completed ones) is or ought to be like a mathematical activity. The issue which motivates and underlies much of the tension in philosophical discussion of computer science is formal verification, or mathematically reasoning about a program's outcome.

The Formal Verification Debate

The use of formal verification in computer science has generated debate since the appearance of a paper on verification and social processes by R. DeMillo, R. Lipton, and A. Perlis in 1979. But it was not until 1988 that these questions drew the attention of a "pure" philosopher, when J. Fetzer resurrected the program verification/social process debate of a decade earlier and subjected it to genuine philosophical analysis. Before this time, debate on the issues was evidenced mainly by differences in visionary accounts of how the young discipline of computer science ought to proceed, given not by philosophers but by computer science practitioners and teachers.

One of the early proponents of formal program verification was John McCarthy, who is also given credit for coining the term "artificial intelligence" in the 1950s. McCarthy was originally motivated by a theory of computation that would allow, among other advantages, the automatic translation from one linguistic paradigm to another. One can, in fact, look back now after nearly thirty years and confirm that automatic program translation, with the help of precise language specification, has been accomplished in the case of language compilers. These are programming tools that translate programs written in familiar human languages like Basic, C++, and Java, into the machine language of computers, which is composed only of zeroes and ones. However, as every language reference manual's warranty disclaimer demonstrates, no automatic compiler in all cases correctly translates programs into machine language. Thus, there is the distinction between (1) using mathematical methods during language translation to produce highly reliable machine language code, and (2) using mathematical methods to prove that a source program would behave, in an abstract sense, exactly as its specification implies. McCarthy, seeing no obstacle to (2), wrote:

> It should be possible almost to eliminate debugging. Debugging is the testing of a program on cases one hopes are typical, until it seems to work. This hope is frequently vain. Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. (McCarthy, 1962, p. 22)

While McCarthy was one of the first to express this opinion, it came to be shared by others, who strove to describe what such a proof would be like. P. Naur recognized that one way to talk about a program both as a static, textual entity and as a dynamic, executing entity, was to conceive of the program as executing, but from time to time to conceptually "halt" it and make statements about the state of its abstract machine at the time of halting. (Naur, 1966) By making a number of these "snapshots" of a conceptually executing program, and by providing justifications for each on the basis of the previous one, a proof about the state of the abstract machine upon termination could be constructed.

Though this idea held much promise for believers in the mathematical paradigm (i.e., that computer science is a branch of formal mathematics), it came under attack in the above-mentioned essay by DeMillo, Lipton, and Perlis. They argued that mechanically produced program verifications, which are long chains of dense logical formulas, are *not* what constitute mathematical proofs. In coming to be accepted, a mathematical proof undergoes social processes in its communication and peer scrutiny, processes that cannot be applied to unfathomable pages of logic. While DeMillo, Lipton, and Perlis did not subscribe to the mathematical paradigm, they also did not deny that programming is *like* mathematics. An analogy can be drawn between mathematics and programming, but "the same social processes that work in mathematical proofs doom verifications." (DeMillo et al., 1979, p. 275) Social processes, they argued, are critical:

> No matter how high the payoff, no one will ever be able to force himself to read the incredibly long, tedious verifications of real-life systems, and unless they can

> be read, understood, and refined, the verifications are worthless. (DeMillo et al., 1979, p. 276)

Although Fetzer was also a critic of the mathematical paradigm for computer science, it was for different reasons. He argued that the presence or absence of social processes is germane to neither the truth of mathematical theorems nor program verifications:

> Indeed, while social processes are crucial in determining what theorems the mathematical community takes to be true and what proofs it takes to be valid, they do not thereby make them true or valid. The absence of similar social processes in determining which programs are correct, accordingly, does not affect which programs are correct. (Fetzer, 1988, p. 1049)

DeMillo, Lipton, and Perlis hit upon, for example, the boredom, tedium, and lack of glamor involved in reviewing proofs produced by mechanical verifiers. But for Fetzer, if this is all there is to their criticism of formal verification, it is not substantial. As Fetzer pointed out, social processes are characterized by transitory patterns of human behavior which, one could imagine, in different circumstances would reserve for program verification the same sort of excitement and collegial collaboration which marks the best mathematical research. Thus DeMillo, Lipton, and Perlis have identified a difference in *practice* between mathematical research and formal program verification, but not in *principle*.

Fetzer believes that formal program verification cannot fulfill the role that some of its advocates would assign to it within software engineering, but he attacks it from a nonsocial, more strictly philosophical perspective. This has to do with the relationship between mathematical models and the causal systems they are intended to describe. Close scrutiny of this relationship reveals, for Fetzer, the relative, rather than absolute, nature of the program correctness guarantee that formal verification can provide. It is only possible to prove formally something about a formal model, that is, a formal program model rendered in formal text. It is not possible to prove formally something about a causal model, that is, an actual, executing program represented in a physical, electronic substrate of bistable processor and memory elements. "[I]t should be apparent that the very idea of the mathematical paradigm for computer science trades on ambiguity." (Fetzer, 1991, p. 209.)

Strong claims of formal verificationists are victim to this ambiguity because they ignore several distinctions: between programs running on abstract machines with *no* physical counterpart and programs running on abstract machines *with* a physical counterpart; between "programs-as-texts" and "programs-as-causes"; and between pure and applied mathematics. Recognizing these distinctions, for Fetzer, reveals that the claim that it is possible to reason in a purely *a priori* manner about the behavior of a program is true if the behavior is merely abstract; false, and dangerously misleading otherwise. This guarantees the indispensability of empirical methods in the software development process, for example, the use of program testing in an effort to eliminate program bugs.

Abstraction in Computer Science

Computer scientists are often thought to labor exclusively in a world of bits, logic circuits and microprocessors. Indeed, the foundational concepts of computer science are described in the language of binary arithmetic and logic gates, but it is a fascinating aspect of the discipline that the *levels of abstraction* that one can lay upon this foundational layer are limitless, and make possible to model familiar objects and processes of every day life entirely within a digital world. When digital models are sufficiently realistic, the environments they inhabit are called virtual worlds. So today, of course, there are virtual libraries, virtual shopping malls, virtual communities, and even virtual persons, like the digital version of actor Alan Alda created in an episode of PBS's Scientific American Frontiers.

Complex virtual worlds such as these are made possible by computer scientists' ability to distance themselves from the mundane and tedious level of bits and processors through tools of abstraction. To abstract is to describe something at a more general level than the level of detail seen from another point of view. For example, an architect may describe a house by specifying the height of the basement foundation, the location of load-bearing walls and partitions, the R-factor of the insulation, the size of the window and door rough openings, and so on. A realtor, however, may describe the same house as having a certain number of square feet, a certain number of bedrooms, whether the bathrooms are full or half, and so on. The realtor's description leaves out architectural detail but describes the same entity at a more general level, and so it is an abstraction of the architect's description. But abstraction is relative. For example, the architect's description is itself an abstraction when compared to a metallurgist's description of the nails, screws, and other fasteners making up the house, and the botanist's description of the various cellular properties of the wood it contains.

The computer scientist's world is a world of nothing but abstractions. It would not be possible to create the complex virtual worlds described above if the only things computer scientists could talk about were bits, bytes, and microcircuits. One can give an accurate idea of what computer scientists do by describing the abstraction tools they use. Now to characterize computer science as involved with abstractions seems to claim for it a place alongside mathematics as a purely formal endeavor. But the general trends in all programming are toward higher-quality software by abstracting away from the lower-level concepts in computer science and toward the objects and information that make up the real world. This is a kind of abstraction that is fundamentally different from that which takes place in mathematics. Understanding the difference is crucial in avoiding the persistent misconception by some that computer science is just a branch of pure mathematics. Both mathematics and computer science are marked by the introduction of abstract objects into the realm of discourse, but they differ fundamentally in the nature of these objects. The difference has to do with the abstraction of *form* versus the abstraction of *content*.

Traditionally, mathematics, as a formal science, has been contrasted with the factual sciences such as physics or biology. As natural sciences, the latter are not concerned with abstraction beyond that offered by mathematics as an analytical tool. The literature is full of

strict bifurcations between the nature of formal and factual science in terms of the meanings of the statements involved in them. R. Carnap, for example, employs the analytic/synthetic distinction in claiming that the formal sciences contain only analytic statements. Since analytic statements are true only by virtue of the transformation rules of the language in which they are made, Carnap is led to the view that "*[t]he formal sciences do not have any objects at all*; they are systems of auxiliary statements without objects and without content." (Carnap, 1953, p. 128) Thus, according to Carnap the abstraction involved in mathematics is one totally away from content and toward the pure form of linguistic transformations.

Not all philosophers of mathematics agree with Carnap that mathematics has only linguistic utility for scientists, but there is agreement on the nature of mathematical abstraction being to remove the meanings of specific terms. M. Cohen and E. Nagel, for example, present a set of axioms for plane geometry; remove all references to points, lines, and planes; and replace them with symbols used merely as variables. They then proceed to demonstrate a number of theorems as consequences of these new axioms, showing that pure deduction in mathematics proceeds with terms that have no observational or sensory meaning. An axiom system may just *happen* to describe physical reality, but that is for experimentation in science to decide. Thus, again, a mathematical or deductive system is abstract by virtue of a complete stepping away from the content of scientific terms:

> Every [deductive] system is of necessity *abstract*: it is the structure of certain *selected* relations, and must consequently omit the structure of other relations. Thus the systems studied in physics do not include the systems explored in biology. Furthermore, as previously shown, a system is deductive not in virtue of the special meanings of its terms, but in virtue of the universal relations between them. The specific quality of the things which the terms denote do not, as such, play any part in the system. Thus the theory of heat takes no account of the unique sensory qualities which heat phenomena display. A deductive system is therefore doubly abstract: it abstracts from the specific qualities of a subject matter, and it selects some relations and neglects others. (Cohen et al., 1953, pp. 138–139)

As a final example, consider C. Hempel's assessment of the nature of mathematics while arguing for the thesis of *logicism*, or the view that mathematics is a branch of logic:

> The propositions of mathematics have, therefore, the same unquestionable certainty which is typical of such propositions as "All bachelors are unmarried," but they also share the complete lack of empirical content which is associated with that certainty: The propositions of mathematics are devoid of all factual content; they convey no information whatever on any empirical subject matter. (Hempel, 1953, p. 159)

In each of these accounts of mathematics, all concern for the content or subject matter of specific terms is abandoned in favor of the *form* of the deductive system. So the abstraction involved results in essentially the *elimination* of content. In computer science,

content is not totally abstracted away in this sense. Rather, abstraction in computer science consists in the *enlargement* of content. For computer scientists, this allows programs and machines to be reasoned about, analyzed, and ultimately efficiently implemented in physical systems. For computer users, this allows useful objects, such as documents, shopping malls, and chat rooms, to exist virtually in a purely electronic space.

Understanding abstraction in computer science requires understanding some of the history of software engineering and hardware development, for it tells a story of an increasing distance between programmers and the machine-oriented entities which provide the foundation of their work. This increasing distance corresponds to a concomitant increase in the reliance on abstract views of the entities with which the discipline is fundamentally concerned. These entities include machine instructions, machine-oriented processes, and machine-oriented data types. The remainder of this chapter will explain the role of abstraction with regard to these kinds of entities.

Language abstraction. At the grossest physical level, a computer process is a series of changes in the state of a machine, where each state is described by the presence or absence of electrical charges in memory and processor elements. But programmers need not be directly concerned with machine states so described, because they can make use of software development tools which allow them to think in other terms. For example, with the move from assembly to high-level language, computer scientists can abandon talk about particular machine-oriented entities like instructions, registers and word integers in favor of more abstract statements and variables. High-level language programs allow machine processes to be described without reference to any particular machine. Thus, specific language content has not been eliminated, as in mathematical or deductive systems, but replaced by abstract descriptions with more expressive power.

Procedural Abstraction. Abstraction of language is but one example of what can be considered the attempt to enlarge the content of what is programmed about. Consider also the practice of *procedural abstraction* that arose with the introduction of high-level languages. Along with the ability to speak about abstract entities like statements and variables, high-level languages introduced the idea of *modularity*, according to which arbitrary blocks of statements gathered into *procedures* could assume the status of statements themselves. For example, consider the high-level language statements given in Figure 1. It would take a studied eye to recognize that these statements describe a process of filling an $n \times m$ matrix A and an $m \times p$ matrix B with numbers and multiplying them, putting the result in an $n \times p$ matrix C such that $C_{i,k} = \sum_{j=1}^{m} A_{i,j} B_{j,k}$. But by abstracting out the three major operations in this process and giving them procedure names, the program can be written at a higher, and more readable, level as in Figure 2. These three statements convey the same information about the overall process, but with less machine detail. No mention is made, say, of the order in which matrix elements are filled, or indeed of matrix subscripts at all. From the point of view of the higher-level process, these details are irrelevant; all that is really necessary to invoke the process is the names of the input and output matrices and their dimensions, given as parameters to the lower-level procedures. Of course, the details of how the lower-level procedures perform their actions must be given in their definitions, but

```
for i ← 1 to n do
    for j ← 1 to m do
        read(A[i,j])
for j ← 1 to m do
    for k ← 1 to p do
        read(B[j,k])
for i ← 1 to n do
    for k ← 1 to p do begin
        C[i,k] ← 0
        for j ← 1 to m do
            C[i,k] ← C[i,k] + A[i,j] * B[j,k]
    end
```

Figure 1: *Multiplying Matrices*

```
ReadMatrix(A,n,m)
ReadMatrix(B,m,p)
MultiplyMatrices(A,B,C,n,m,p)
```

Figure 2: *Multiplying Matrices with Procedural Abstraction*

the point is that these definitions can be strictly separated from the processes that call upon them. What remains, then, is the total abstraction of a procedure's use from its definition. Whereas the language example had the abstraction of the content of computer instructions, here there is the abstraction of the content of whole computational procedures. And again, the abstraction step does not eliminate content in favor of form as in mathematics; it renders the content more expressive.

Data Abstraction. As a last example, consider the programmer's practice of *data abstraction.* Machine-oriented data types, such as integers, arrays, floating point numbers, and characters, are, of course, themselves abstractions placed on the physical states of memory elements interpreted as binary numbers. They are, however, intimately tied to particular machine architectures in that there are machine instructions specifically designed to operate on them. (For example, integer instructions on one machine may operate on 32 bits while similar operations on another machine may operate on 64 bits.) They are also built into the terminology of all useful high-level languages. But this terminology turns out to be extremely impoverished if the kinds of things in the world being programmed about include, as most current software applications do, objects like customers, recipes, flight plans, or chat rooms.

The practice of data abstraction is the specification of objects such as these and all operations that can be performed on them, without reference to the details of their implementation in terms of other data types. Such objects, called *abstract data types* (ADTs), once they become implemented, assume their place among integers, arrays, and so on as legitimate objects in the computational world, with their representation details, which are necessarily more machine-oriented, being invisible to their users. The result is that programs that are about customers, recipes, flight plans, and so on are written in terms that are natural to these contexts, and not in the inflexible terms of the underlying machine. The programs are therefore easier to write, read, and modify. The specification and construction of abstract data types are primary topics in undergraduate computer science curricula, as evidenced by the many textbooks devoted to these topics. But this again is a type of abstraction that does not eliminate empirical content, as in mathematics, but rather enlarges the content of terms by bringing them to bear directly on things in a nonmachine-oriented world.

Conclusion

Computer science will always be built on a scientific and engineering foundation requiring specialists with the most acutely analytic, creative, and technological minds. But the modeling and abstraction abilities that this foundation provides opens the field to virtually anyone willing to learn its languages. As computer science grows as a discipline, its methodology will be less dependent on the specialists maintaining the foundation and more dependent on those able to develop, implement, and use high-level languages for describing computational processes. What is meant by a "computational process" has diverged so much from the notion of a machine process that a currently popular language paradigm, namely object-oriented design and programming, deemphasizes traditional algorithmic forms of program control in favor of the notions of classes, objects, and methods. (See, for example, the current widespread interest in Java.)

As programming languages evolve, it will be necessary for software developers to be conversant in the analytical tools of philosophers as they analyze their domains for logics, rules, classifications, hierarchies, and other convenient abstractions. Much of the computational modeling process will be characterized by activity more akin to logic and ontology than programming *per sé*. (See the chapters in this volume by Smith, Antonelli, Gillies, and Bicchieri on ontology, logic, and probability.) Now more than ever, there is room for much philosophical research in the development of future computational modeling languages. One might even venture the prediction that philosophy will come to be an influential tool in the analysis and practice of computer science methodology.

References

Carnap, R. (1953). Formal and factual science. In H. Feigl and M. Brodbeck (eds.) Readings in the philosophy of science (pp. 123–128). New York: Appleton-Century-Crofts. Carnap's

paper is one of a group of seminal papers on the philosophy of mathematics collected in this anthology. They are central to understanding the distinction between mathematics and science, and the role mathematics plays in science. For the advanced philosophy student.

Cohen, M., and Nagel, E. (1953). The Nature of a logical or mathematical system. In H. Feigl and M. Brodbeck (eds.) Readings in the philosophy of science (pp. 129–147). New York: Appleton-Century-Crofts. This paper is from the same group as (Carnap, 1953).

DeMillo, R., Lipton, R., and Perlis, A. (1979). Social processes and proofs of theorems and programs. Communications of the ACM, 22, 271–280. This paper ignited a debate within the computer science community regarding the role of formal methods in computer science methodology, specifically program verification. For the intermediate computer science student.

Fetzer, J. (1988). Program verification: the very idea. Communications of the ACM, 31, 1048–1063. This paper rekindled the program verification debate from a philosopher's point of view and caused impassioned responses from the computer science community. For the intermediate philosophy student.

Fetzer, J. (1991). Philosophical aspects of program verification. Minds and Machines, 1, 197–216. This paper summarized the wreckage of the program verification debate, again from the philosophical point of view, arguing against the mathematical paradigm. For the intermediate philosophy student.

Floyd, C. (1987). Outline of a paradigm change in software engineering. In G. Bjerknes et al (Eds.) Computers and democracy: a Scandinavian challenge (pp. 191–210). Hants, England: Gower. This paper was one of the first to come out of the computer science community advocating a view of software as process rather than product. For the introductory student in any discipline.

Hailperin, M., Kaiser, B., and Knight, K. (1999). Concrete abstractions: an introduction to computer science. Pacific Grove, CA: PWS Publishing. An excellent introduction to computer science that emphasizes the role of abstractions in computer science methodology. For the first time computer science student.

Hempel, C. (1953). On the nature of mathematical truth. In H. Feigl and M. Brodbeck (eds.) Readings in the philosophy of science (pp. 148–162). New York: Appleton-Century-Crofts. This paper is from the same group as (Carnap, 1953).

Hoare, C. (1986). Mathematics of programming. BYTE, August, 115–149. A clear statement of the view that computer science is a species of pure mathematics. For the advanced

computer science student.

McCarthy, J. (1962). Towards a mathematical science of computation. Proceedings of the IFIP Congress, 62, 21–28. Another statement of the mathematical paradigm for computer science, with an emphasis on the role of recursion in computer science methodology. For the advanced computer science student.

Naur, P. (1966). Proof of algorithms by general snapshots. BIT, 6, 310–316. An example of mathematical reasoning about programs in pursuit of program verification. For the advanced computer science student.

Further Reading

Colburn, T. (2000). Philosophy and computer science. Armonk, NY: M.E. Sharpe. Concerns the philosophical foundations of computer science and the contributions that philosophy and computer science can make to each other. Part 3 in particular concerns philosophical aspects of computer science methodology. For the introductory philosophy student.

Colburn, T., Fetzer, J., and Rankin, T. (Eds). (1993). Program verification: fundamental issues in computer science. Studies in Cognitive Systems. Dordrecht, Netherlands: Kluwer Academic Publishers. An anthology of papers centering around the role of formal methods in computer science. Difficulty level varies.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). Introduction to algorithms. Second edition. Cambridge, MA: The MIT Press. The definitive undergraduate text on algorithms and data structures. For the advanced computer science student.

Gumb, R. (1989). Programming logics: an introduction to verification and semantics. New York: John Wiley and Sons. An example of the focus on program verification from a formal mathematical point of view. For the advanced computer science student.

Myers, G. (1979). The Art of software testing. New York: John Wiley and Sons. An example of the focus on program verification from a program testing point of view. For the intermediate computer science student.

Pfleeger, S. (2001). Software engineering: theory and practice. 2d ed. Upper Saddle River, New Jersey: Prentice Hall. An example of the current practice in software engineering methodology. For the advanced computer science student.

Signature

TIMOTHY R. COLBURN
Timothy Colburn has been professor of computer science at the University of Minnesota-Duluth since 1988. Prior to that, he was principal research scientist for Honeywell, Inc. He is the author of *Philosophy and Computer Science* (Explorations in Philosophy Series, M.E. Sharpe, 2000) and co-editor of *Program Verification: Fundamental Issues in Computer Science* (Studies in Cognitive Systems Series, Kluwer, 1993).

Glossary

ABSTRACT DATA TYPE See DATA ABSTRACTION.

ABSTRACTION To describe something at a more general level than the level of detail seen from another point of view. For example, a realtor's description of a house may leave out architectural detail so that it is an abstraction of an architect's description. In computer science, language abstraction is the use of high-level language programs allowing computational processes to be described without reference to any particular machine.

ALGORITHM Any well-defined sequence of steps that takes some value as input and produces some value as output. (Cormen et al, 2001, p. 5)

COMPILER A programming tool that translates a program written in a familiar high-level language like Basic, C++, or Java, into, typically, the machine language of a computer, which is composed only of zeroes and ones.

COMPUTATIONAL PROCESS A process distinguished from a physical process, in that it is studied in ways that ignore its physical nature. (Hailperin et al, 1999, p. 3) For example, the process by which a card player arranges cards in her hand, and the process by which a computer sorts names in a customer list, though they share nothing in common physically, may nevertheless embody the same computational process.

COMPUTER SCIENCE The science concerned with the study of computational processes.

DATA ABSTRACTION The specification of computational objects (like customers, recipes, flight plans, chat rooms, etc.) and all operations that can be performed on them, without reference to the details of their implementation in terms of other data types. Such objects, called *abstract data types,* once they become implemented, assume their place among integers, arrays, and so on as legitimate objects in the computational world, with their representation details, which are necessarily more machine-oriented, being invisible to their users.

DATA STRUCTURE A way to store and organize data in order to facilitate access and modifications. (Cormen et al, 2001, p. 8)

DEBUGGING The process of eliminating errors (or "bugs") from a computer program.

FORMAL PROGRAM VERIFICATION The process of determining whether a program conforms to its specification, not by empirically testing the program to observe its behavior, but by mathematically reasoning about its algorithm as a formal, abstract object.

FORMAL PROGRAM VERIFICATION DEBATE The debate over whether formal program verification can offer guarantees of program correctness and reliability. Opponents of formal program verification claim that it can offer no such guarantees. Proponents believe that it can largely replace program testing in the identification of program bugs (errors).

LOGICAL POSITIVISM The view advocating the reduction of theoretical vocabulary to observational vocabulary.

LOGICISM The view that all of mathematics can be reduced to logic.

OBJECT-ORIENTED PROGRAMMING A currently popular programming paradigm, based on the principles of data abstraction, that deemphasizes traditional algorithmic forms of program control in favor of the notions of classes, objects, and methods.

PROCEDURAL ABSTRACTION The separation of a computational procedure A's use from its definition, so that another procedure B can call A to get something accomplished while remaining ignorant of how A accomplishes its task.

PROGRAM SPECIFICATION A detailed description of a computer program's input and output, ignoring the details of how the program actually accomplishes its task.

PROGRAM VERIFICATION The process of determining whether a program conforms to its specification.