(Re)Defining Computing Curricula by (Re)Defining Computing

Charles L. Isbell
Georgia Institute of Technology
isbell@cc.gatech.edu

Lynn Andrea Stein
Olin College
las@olin.edu

Robb Cutler
Purdue University and Computer Science Teacher's Association
robb.cutler@purdue.edu

Jeffrey Forbes
Duke University
forbes@cs.duke.edu

Linda Fraser
Waiariki Institute of Technology
Linda.Fraser@waiariki.ac.nz

John Impagliazzo
Qatar University
John@qu.edu.qa

Viera Proulx
Northeastern University
vkp@ccs.neu.edu

Steve Russ
University of Warwick
sbr@dcs.warwick.ac.uk

Richard Thomas
Queensland University of Technology
r.thomas@qut.edu.au

Yan Xu
Microsoft
yanxu@microsoft.com

ABSTRACT
What is the core of Computing? This paper defines the discipline of computing as centered around the notion of *modeling,* especially those models that are automatable and automatically manipulable. We argue that this central idea crucially connects models with languages and machines rather than focusing on and around computational artifacts, and that it admits a very broad set of fields while still distinguishing the discipline from mathematics, engineering and science. The resulting computational curriculum focuses on modeling, scales and limits, simulation, abstraction, and automation as key components of a computationalist mindset.

Categories and Subject Descriptors
K.3.2 [**Computer and Education**]: Computer and Information Science Education – *curriculum.*

General Terms
Standardization.

Keywords
Computing, Computationalist Mindset, Computational Thinking.

1. Introduction

In this paper, we present the carefully considered opinions of a diverse group of academics, from the fields that comprise computing, on the question of our discipline's core and on how aspects of that core should be generally understood by informed citizens as well as by those who practice computing in various ways.

We take the position that computing is a discipline unto itself—neither math nor science nor engineering nor anything else, though it overlaps with many of these—and that it is distinguished by a mindset that we call *computationalist thinking*. Here, we use the term computationalist merely to mean *someone who does computing*, and nothing more nor less.[1]

As a discipline, computing brings together models, languages, and machines to represent and generate processes. The heart of computing is not the particular artifacts around which our curricula often revolve. Instead, this key idea—that models, languages, and machines are equivalent—is the fundamental core of computing. Further, this idea admits a broad set of practices and specialities, including computer science, information science, human-centered computing, software engineering, and many others, as well as what we will call more generally contextualized computing.

From this position, we also argue that the curricula of existing courses should be revisited to inculcate the computationalist mindset—specifically, core competencies in modeling, scales and limits, simulation, abstraction, automation, and interpretation of data. For core computationalists for whom the historical computing curriculum centers on understanding or using the machine, we propose that courses also include a focus on models and languages—the intellectual frameworks of computationalist thinking. For contextualized computationalists, curricula grounded in principles of computationalist thinking tailored to domain-specific needs has the potential to be transformative, not only by encouraging innovation within a domain but also by creating entirely new disciplines. Lastly, at both the secondary and post-secondary levels, we urge that a minimal level of computationalist literacy be required of all students. In some cases, this may require the design of completely new courses; in other cases, the organizational structure of existing courses can be adapted.

In the next sections, we provide background and context for this report, motivating the need to address these issues now. We then explain our notion of computing-in-the-large as bringing models, languages, and machines together, carefully defining our terms at that point. We next show how a variety of fields fit into our definition before finally turning to curricula and discussing what learning outcomes should be integrated into our educatioal sys-

ITiCSE'09, July 6-9, 2009, Paris, France.

[1] We prefer our philosophical fights to be about computing itself, not the word we use to denote those who do computing; however, we recognize that the term may generate controversy regardless of our disclaimers, as would just about any term. We hope that the reader will trust our intent, and try not to imbue computationalist with any undue meaning or infer that we are making a larger point by its use.

tems for different audiences of learners. We conclude by making recommendations for next steps in extending what we have presented here and in presenting these ideas to the larger community.

2. Background of this Report

This document reports on the activities and conclusions of a Working Group convened at the Conference on Innovation and Technology in Computer Science Education at the Université Pierre et Marie Curie (Paris VI) during July, 2009 on the question of our discipline's core. It does not purport to be a consensus of the discipline, representative of all constituencies or opinions, or a strict work of scholarship. It is informed by scholarship—many of the participants have engaged in various studies of the discipline, of educational best practices, of industrial and academic needs—and it is firmly grounded in our own experiences as well as an emerging shared understanding of the changes that our field has undergone. We offer it as a perspective for general consideration, much in the spirit of Wing [19], Denning [8] and others.

The ten members of the working group came from around the world (but largely from anglophone educational systems) and included representation from large research universities and small teaching colleges, liberal arts traditions and engineering-focused institutions, as well as secondary education and the computer industry. Most of us were trained in what are broadly considered core computing fields, though many have worked at the interface between computing and other disciplines. All have had a hand in curriculum design, development, and reform; some began careers with this focus, while others have come to these questions after significant engagement in computational research or development activities.

We came together motivated by a sense that computing as typically taught is too much about the computer and not enough about what we have come to call the computationalist mindset; that the kind of thinking that comes from this mindset—centered on a particular kind of modeling that allows automation, simulation, exploration, as well as clarification of the original problem domain—is critical to a variety of emerging fields; and that the educated person in our societies needs a basic facility with these intellectual tools.

Our goals in making this report are straightforward: We want to define a broad field of computing; understand the role of currently recognized sub disciplines in that field; identify what is important and should be generally understood by informed citizens and those who will practice computing in various ways; and begin to make recommendations on how to communicate these important ideas to students. In particular, we want to articulate the fundamental properties of computing and the computationalist mindset, in order to understand what computing curricula built around this way of thinking might look like.

3. Why Worry About This Now?

We believe that computing education is in crisis. Curricula have grown too large for many institutions and programs. Several adjoining disciplines—information science, software engineering, computer engineering, information technology, and informatics—as well as computer science make compelling claims on computing, incorporating different but equally voluminous material (such as [15]). There is increasing interest in *computational X* degrees: programs bridging computing and other disciplines such as biology, social science, art, and economics [3]. Some suggest universal computing literacy in the form of computer programming [10]; others advocate a focus on our mechanics or on modes of thinking [8], [19]. Computing has become an inter- and intra- disciplinary field of intertwined concepts pervading not just most technical fields, but society at large [1].

Much of this is to the good; however, it has become increasingly difficult to understand how to teach computing. Unlike subjects such as mathematics and reading, which benefit from a long history of pedagogical research into learning and assessment, computing is a relatively young discipline and one with a reputation for being poorly taught. Indeed, the field of computing is itself struggling for an identity even as academic institutions are trying to determine what should be taught, when and how. In order to cause a dramatic change in computing education, we need to create a model that clarifies what the field is, and why and how to study it. We need to make those reasons clear to our students and their parents, to professionals, and to ourselves.

The most recent US curricular standard recognizes five areas of computing: computer engineering, computer science, information systems, information technology, and software engineering [15]. These areas overlap but for the most part focus inwards, on the hardware, software, and systems of traditional computation.[2] Often, the computer science curriculum in particular amounts to a history of our artifacts—the machines, the system components, the applications—rather than an explication of the key ideas at the core of our discipline (see, e.g., [7]).

Some of the controversy that inevitably arises from these sorts of discussions comes from the fact that computers—the artifacts our discipline enables, uses, and in some cases studies—are uniquely compelling devices. In some cases, what is taught as computer science is actually computer use or computer literacy; this is particularly prevalent among pre-college curricula [18]. Even when the broad range of computing disciplines at the post-secondary level is considered, it can sometimes be easy to fall into the trap of believing that computing is about electronic computers.

Yet another confusion is the result of beginning with an assumption that computing must be either mathematics, or science, or engineering. This is unsurprising because it is easy to draw those connections. Like mathematics, we build models; unlike mathematics', our models are active and effect-making: they cause things to happen. Like science, we study a system that exists in nature; however, like engineering, our systems are artificial technology and subject to complex trade-offs in implementation. Computing also bears resemblance to the arts—the creation of artifacts—to humanities—the study of texts—and to the social sciences—the study of humans and societies.

A number of curricular reforms have tried to confront these challenges. At Georgia Tech, the Threads curriculum has identified eight overlapping ways of being a computationalist; each student follows at least two of these paths, but no student completes all

[2] The two fields with "information" in their names are the least prone to this inward-focused tendency and, perhaps as a result, the ones most likely to be omitted from conversations about computing. Their inclusion in the computing core is a first step in the direction for which we wish to argue.

eight during an undergraduate career [12]. Olin's "small footprint" curriculum for computing creates a reduced core focused on the key approaches and concepts necessary to learn the rest of computing [9]. Union College, Georgia Tech, and The College of New Jersey are among the institutions offering both departmental introductions and follow-on programs in digital media computation [13], [20], and computation is increasingly being understood as critical to scientists [21].

Universities have also begun to reorganize themselves around notions of computing as a discipline by creating academic units of computing at the same level corresponding to academic units of engineering or science. The Bren School at UC Irvine has three overlapping degrees representing three different approaches to computing and is housed in a unit reporting directly to their Provost; Georgia Tech has a College of Computing offering multiple computing degrees with a Dean that also reports to their Provost. University of Michigan has an Information School with a Dean reporting directly to their chief academic officer.

Even so, we are still faced with a difficult problem of understanding how to convey the core of computing in a variety of different contexts and with limited time. We will suggest below that the curriculum of existing courses be revisited to inculcate what we call computationalist thinking—specifcally, core competencies in modeling, scales and limits, simulation, abstraction, and automation. In some cases, this may require the design of completely new courses; in other cases, the organizational structure of existing courses can be adapted. But first, we begin by examining several existing visions of what that core might be.

4. Visions of Computing

In saying that computing is a singular discipline, we are suggesting that there are certain ways of thinking that are characteristic of all computationalists, including those whose primary concerns are the marriage of computational disciplines with other fields. These ways of thinking are shared by members of the five subdisciplines identified by CC2005 as well as by computational media practitioners, bioinformaticists, quantitative social scientists, and others we call contextualized computationalists. We believe that the most basic principles of this mindset should also be shared by all educated persons.

Wing [19] takes a similar stance, calling this way of thinking *computational thinking*.[3] Wing suggests that it includes: seeking algorithmic approaches to problem domains; a readiness to move between differing levels of abstraction and representation; and familiarity with decomposition, separation of concerns and modularity. We find much to like in Wing's approach but would shift emphasis from algorithm to interaction—less about finding answers and more about providing services, interfaces, behaviors—and would highlight our fusion of models, languages, and machines in what we call computationalist thinking. In particular, we will advocate below for a more central role for the activity of modeling and would consequently add: a readiness to adopt a deliberate modeling approach to phenomena where we identify features of a domain that are relevant to our interests, formulate relationships between those features and identify the relevant agencies that are sources of change in the domain.[4]

Denning [8] takes a more pragmatic approach to the computational core, which he divides into mechanics, design, and practices. His interest is in the generalizable principles of each. Denning's notion of mechanics is inspired by that subdiscipline of physics and includes computation, communication, coordination, automation, and recollection. Each of these activities has a role to play in our world of computationalist thinking. Denning's design principles include simplicity, performance, reliability, evolvability, and security; several of these are a part of our notion of scale and limits. Computational practices, according to Denning, include programming, engineering systems, modeling and validation, innovating, and applying. Many computationalists engage in these practices while adhering to these design principles and observing these mechanics, as we acknowledge in section 6.1; however, our notion of the computationalist mindset is at a more abstract level, less tied to the particular subsets of these lists that typify individual subdisciplines, and our vision of the shared computationalist core is correspondingly less tied to actual computer systems.

In Reflections from the Field [14], the Computer Science and Telecommunications Board describe the core activities of Computer Science (rather than the broader field of computing). According to this report, "Computer Science involves the creation and manipulation of abstractions [and] the creation and study of algorithms, ... deals with artificial constructs notably unlimited by physical laws, ... exploits and addresses exponential growth, ... studies fundamental limits on what can be computed, and ... addresses the complex, analytic, rational action that is associated with human intelligence." Again, we agree that many computationalists engage in these activities, but we seek a core mindset that is shared by different kinds of computationalists—not just computer scientists—and that further is of benefit to those whose computationalism may be more contextualized or even a matter of basic literacy.

The report of the Northeast Workshop on Integrative Computing Education and Research [3] specifically addresses the role of mindset, especially for contextualized computation:

> Our greatest contribution to integrating computer science with other disciplines will be our unique mindset: our conceptual base, our style of reasoning, and our values... Computational paradigms have changed the core of many disciplines and enabled new kinds of questions.

That is, it is the way that computationalists think—how we approach the world—that underlies our greatest contributions to neighboring disciplines. While this mindset produced the computational artifacts that are changing our world, it should not be confused with them. In the future, our artifacts will be different, but the core computationalist mindset that we lay out below should remain.

There are, of course, many other visions of computing and its

[3] In this document, we have used the terms computationalist mindset and computationalist thinking instead as a means of indicating that we mean only the mindset or way of thinking of computationalists (that is, those who do computing) without commentary on other similar terms with their own specific meanings.

[4] Terms like *algorithmic thinking* might then be viewed as a special case of computationalist thinking and modeling where we focus attention on certain reliable patterns of execution.

related disciplines. Abelson and Sussman [1] famously state that computer science is neither about computers nor a science. Instead, they argue that it is "no more (and no less) than the discipline of constructing appropriate descriptive languages." Foley [11] builds on this notion to suggest that computer science marries this process knowledge—"how to"—with a concern for organization. Felleisen and Krishnamurthy [10], in contrast, argue that what is crucial about computing—at least for broad literacy—is a kind of "imaginative programming" that closely aligns with mathematics, bringing it to life. According to Felleisen and Krishnamurthy, programming "is our field's single most valuable skill."

In the next sections, we will articulate and detail our own particular vision of what computing is. It is not the same as the visions expressed above, but we believe our vision shares much in common with many of them and, in particular, is a reasonable effort at trying to capture something both fundamental and broad about computationalist thinking.

5. Models, Languages, and Machines

In our view, computing is fundamentally a *modeling* activity. Any modeler must establish a correspondence between one domain and another. For the computational modeler, one domain is typically a phenomenon in the world or in our imagination while the other is typically a computing *machine,* whether abstract or physical. The computing machine or artifact is typically manipulated through some *language* that provides a combination of symbolic representation of the features, objects, and states of interest as well as a visualization of transformations and interactions that can be directly compared and aligned with those in the world. The centrality of the machine makes computing models inherently executable or automatically manipulable and, in part, distinguishes computing from mathematics. Therefore, the computationalist acts as an intermediary between *models, machines,* and *languages* and prescribes objects, states, and processes.

These three words—*model, language,* and *machine*—will mean different things to different readers, so we should take a moment to explain further the ways in which we use these terms.

We have a liberal view of machines. Those who study human-centered computing, for example, include humans as a crucial part of the machine system. In our view, **a machine is simply a physical entity capable of carrying out work (including computational work) in the world**.

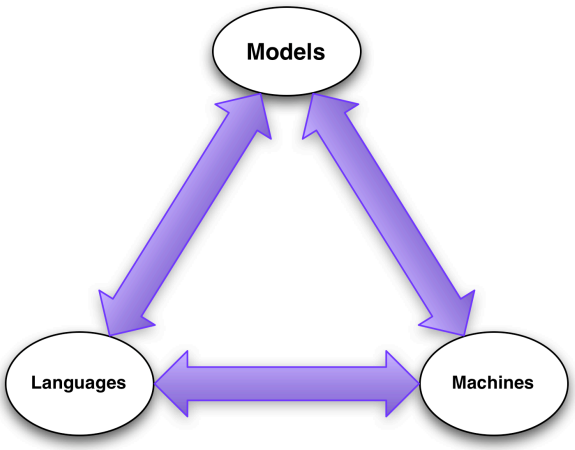

Figure 1. The Tight Coupling of Models, Languages and Machines. The computationalist is a modeler who uses languages to specify machines.

When we say model, all that we mean is a *representation* of some information, physical reality, or a virtual entity in a manner that can then be interpreted, manipulated, and transformed. A model allows one to manipulate and simulate in a way that is easier than the phenomenon modeled—or at least has useful additional affordances—while still retaining crucial predictive or causal powers. **In particular, a computational model:**

- **provides the ability to manipulate and simulate, while spanning volume, distance, and time, and allowing hypotheticals;**
- **retains the ability to cause change in the actual world of the things being modeled;**
- **and hides details and aspects that are not critical to the problem at hand.**

A language allows us to describe in a formal way the process that manipulates our models and transforms them into new models or enables the interpretation of some model in a new way. The language may be Turing complete or a small language or even a simple protocol. Increasingly, computationalists work with "little languages", domain-specific constructs that fit the conceptual space within which they work. What is important about **a language** is that it ***enables* our reasoning and manipulation of the model**.

To close the loop, a machine is any artifact that is capable of accessing these models and performing the transformation processes that are defined in the languages. The machine may be real or it may be virtual or abstract. The key is that **a computational machine allows us to *execute* our models.**

In other words, in computing, our models are languages which are themselves machines; that is, a computational model is manipulable and executable automatically. It can act or can be acted upon. Our languages are models. Our models are executable. Our machines are languages that can themselves be manipulated. It's turtles all the way down.

6. Computing

We can now define **computing as:** ***any purposeful activity that marries the representation of some dynamic domain with the representation of some dynamic machine that provides theoretical, empirical or practical understanding of that domain or that machine.*** Often but not always, computationalists then further actualize those representations by executing them on a physical computing artifact (see Figure 2).

In this way, the practitioners we often call computer scientists and those we often call information scientists are both engaging in computing. In fact, our definition might well encompass parts of operations research, business processes, and even sociology or ethnography, *provided that the results are automatable or automatically manipulable models*.

Some computationalists build running models, or machines. Others construct intellectual models that are more abstract than concrete. Some computationalists focus more on understanding the machines, others on the domains. In any case, every computation-

alist's intellectual toolkit includes both the activity of automation and the ways of thinking—disassembling domains, carving them at their joints—that make this automation possible. Computationalists often build models of processes including concurrent, distributed, and human processes.

Much of the benefit of the computationalist mindset comes from this activity of fitting the model and the language to the needs of the domain. When computationalist thinking met biology, the transformation changed the language that biologists use to describe their own artifacts. Computationalist thinking requires a precision and a disambiguation that is clarifying for the domain to be modeled. Computationalists become proficient in crafting intellectual segmentations of domains that themselves can be significant contributions. But the full impact of computationalist approaches comes when the automation power of our artifacts can be combined with our intellectual tools.

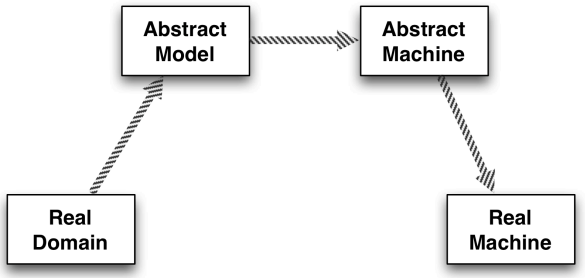

Figure 2. Computationalists Create Automatable Models. Computationalists build models that capture processes. Those process are in turn automatable and automatically manipulable. The computationalist need not engage in each step of this diagram, but the computationalist is engaged in an enterprise that makes each step applicable and possible.

6.1 The Practice of Computing

We know that computing is not only a set of professions but also a single coherent discipline. Just as importantly, it is a widespread practice and a general way of thinking. Computationalists have a systematic body of knowledge they learn and from which they draw; a set of skills and tools they use to practice and apply their knowledge; and a way of thinking and seeing problems that allow them to extend the larger body of knowledge and add value.

Computationalists deal with some or all of the following:

- The computer itself as a technology or technological artifact, as well as a wide range of computing devices
- The problems that computing devices can solve in the abstract unmoored to a specific domain
- The techniques and technologies that enable computational solutions, including the practices that best support these solutions
- The relationship of these tools, techniques, technologies to domains or users, in general
- The study of all of the above using computational tools and mindsets.
- The construction (the science of the construction, the best practices of the construction, the science of the practice of the construction, and so on) of systems using all of the above
- The historical artifacts that have been crucial in the development of these tools and technologies and the future improvement of those artifacts.

It is difficult to characterize precisely the extent of the computing disciplines. Clearly, the core disciplines identified in CC2005 are included within computing. There is also significant overlap among these and with emerging disciplines containing the word computational in their titles (computational biology, chemistry, and physics; computational mathematics; computational media or sometimes digital media; bioinformatics; information science; quantitative social sciences). What characterizes all of these disciplines? What unifies them as computational? We assert that here are two key aspects of these disciplines that makes them computational, and we further suggest that the emphasis is typically placed on the wrong one:

- Computation as device—the machine—changes the scale, scope, and reach of every discipline it touches.
- Computation as mindset causes a reconceptualization of the discipline. The most common symptom of this is what are sometimes called "little languages" [5], special purpose (domain-specific) languages that allow automated manipulation of the domain. The computationalist brings an ability to identify the appropriate abstractions, hide the unnecessary details, and get at the heart of a key process within the domain.

It follows from our theme of computing as a modeling activity that the skills and tools associated with the entirety of computing should be those useful for the construction and management of models. These will include all the familiar skills and tools used in, for example, programming in all its styles, such as integrated development environments, versioning and testing tools, but also include some of those skills and tools used more widely in other modeling disciplines.

6.2 When One Discipline Meets Another

To understand the impact of this thinking and this practice, it is useful to consider a simple case study. For example when computing meets biology it is transformative: one can create a simple model that captures key aspects of behavior (DNA coding, possible manipulations, etc.) and then create a language that describes the interactions and processes possible. Because computational models are executable, computationalist disciplines can scale dramatically, operating on data sets heretofore unimaginable. Because computataionlist disciplines can manipulate huge datasets, the kinds of questions that can be asked are also dramatically transformed. The entire field of precision medicine follows from computation's meeting with biology. Computation's contribution to biology is not so much the processing of large volumes but the two dramatic shifts in thinking that the meeting of these fields created: the reconceptualization that enables automated processing and the reconceptualization which that processing in turn enables.

Computationalists can use non-computers to do computing. Sticking with Biology, for example, the computationalist understands that one can do computing with a vial of saline solution and the components of DNA. In early 1994, Adelman [2] solved a traveling salesman problem by: creating sequenced DNA strands representing cities and complementary strands representing particular streets connecting pairs of cities; placing a few grams of every DNA city and street into a test tube; allowing the natural bonding tendencies of the DNA building blocks to occur; and eliminating strands that could not be valid solutions. Adelman acted as a true computational modeler, creating a representation of cities and streets and creating a correspondence to a representation of the properties of DNA and the processes of chemistry. Finally, he actualized those representations by executing them on the physical realization of his model.

The biologist performing an experiment with DNA in a test tube is not necessarily doing computing; however, the computationalist who sees chemistry as a process manipulating the representation that is DNA can do computing. At the same time, computationalist thinking also allows the biologist to think of cells as machines performing computation, to see certain protein interactions as executing if-then statements or storing and retrieving state. Thus, computing can both use physical processes as tools to do general computing and explain natural processes as doing a specific set of computations, transforming our understanding of what those processes are doing as well as what they can do. Such thinking brings us not only DNA as computing, but manipulable music, participatory art, and other fundamental rethinking of other domains.

7. Who Should Learn Computing and What Should They Learn?

In the process of (re)defining computing, we define three distinct audiences of students. Within each group, our objective is to identify specific and appropriate learning outcomes through the articulation of computing curricula. Figure 3 captures the overlapping nature of the three audiences and their relationship to the entire field of computing.

Our first group contains the pure or core computationalists. These students specialize either in fields such as computer science, software engineering, computer engineering, information science, and information technology, or in disciplines such as multimedia computation where computing is a central focus. While individuals in this group may have some domain-specific knowledge beyond what we might think of as central computing, their overall goal is the deep study of computing rather than the study of any particular domain.

Our second group contains *contextualized* computationalists. These are students who require in-depth knowledge and understanding of particular aspects of computing, but only as they apply to a particular domain. Examples include students of bioinformatics, computational economics, and technical management. In each case, the core field of study is not computing but rather a non-computing discipline strongly shaped and influenced by computing principles.

Our last group includes everyone else. There has been a traditional core curriculum in schools that has existed for a long time. Existing disciplines have expanded and reduced along with society's needs and attitudes. For example in the study of literature, new classics have emerged and in Biology new discoveries have been made; however, we have not universally made room within this core for an entirely new discipline such as computing, even though computers and computerized gadgets and machinery are ubiquitous. We assert that a broad overview of computing knowledge is a fundamental component of being an educated person in the 21st century. Every person will have to interface with computing in many areas of their lives and would, we assert, have a more fulfilling, competent, knowledgeable and self reliant adult life with a good foundation of computing knowledge and skills, Our goal is not to instill specific computing proficiency, but to provide a core computing context for all students. We see this as analogous to the idea that every student should have an understanding of basic scientific principles, have historical perspective, and have read some part of the basic canon of literature.

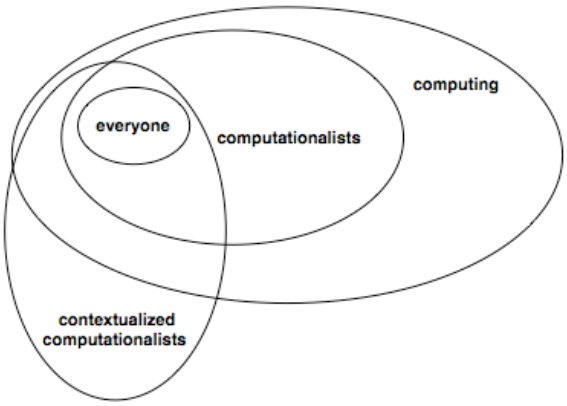

Figure 3. Computing Knowledge Needed by Different Audiences. We believe that everyone should have some working knowledge of computationalist ideas. Contextualized computationalists will generally need to know much more. Note that there are several different kinds of contextualized computationalists who will need more or less computing knowledge and will focus or more or less non-computing knowledge. Computationalists themselves will typically know the most, but the field of computing itself is much too broad for any particular computationalist to be facile in all of its knowledge and tools; nonetheless, there are several core ideas that most computationalists should know.

In fact, each of these groups actually lies on the same spectrum. Conceptually, we are all contextualized computationalists. Many core computationalists tend to be near one side of the spectrum, where the context is a computing machine (or abstraction thereof) itself. As we move slightly along the spectrum we pick up more subareas of computing, for example those who begin to contextualize their efforts more and more by focusing on humans or human processes as a part of the system. Some subareas of human-centered computing, machine learning, artificial intelligence and information science are here, but are no less computing for their broadened interest. Eventually we move far enough along that we begin to start thinking of the focus as being as much about the

domain as it is computational. Some computational X areas such as computational biology begin to appear. Eventually, the domain begins to dominate; we begin to see biologists who use computing devices centrally in their work and may apply the results of computationalist thinking, but do not necessarily try to innovate by finding new ways to apply that thinking. Eventually, we find ourselves at just computer users and then everyone else.

In any case, everyone along the spectrum is well-served by becoming computationalist thinkers or at least being compurate[5]. In fact, we argue that being compurate is as important as being numerate and literate; being illiterate, innumerate, or incompurate is simply not an option in today's world (see Figure 4).

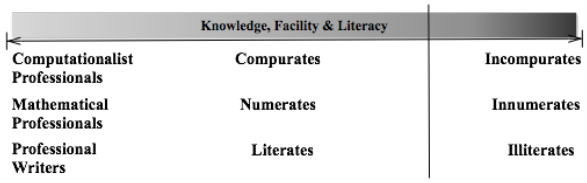

Figure 4. The importance of Being Compurate. At this point, being compurate—having the ability to understand computationalist thinking—is as important as being literate or numerate. Being on the left side of the dividing line is increasingly necessary for the educated person to participate fully in the economic life of most modern societies.

7.1 What We Can Know about Computing

It is beyond the scope of this document to address a full set of learning objectives or outcomes for each audience we have identified above. Still it is worth articulating some of the key ideas that make up computing and identifying how those key ideas are important to each of our groups.

The ideas and thinking involved in fruitfully pursuing these activities more subtle and wide-ranging, and lie at the core of computing. At the very least, computationalist thinking is focused on:

- **Models:** What can computational models afford (make easy)? What do they hide? How do these models relate to specific problems in which the computationalist is engaged? How can such models be manipulated and understood?
- **Abstraction:** How does one effectively distinguish the important aspects of a domain, and for what purpose? How are these aspects realized in a model and executed?
- **Interpretation**: What is the data a particular system manipulates? How does it interpret that data? What is the language of manipulation? For example, the same bit pattern can mean 65 or "A" or *true*. It is in the model and the interpretation that we know the answer.
- **Scales and Limits:** How are classes of problems related to one another by their complexity? How can we usefully distinguish between them? Where does the dependence on inputs lie? What are the tradeoffs of space and time? Why are some problems hard? What cannot be solved exactly? What can be approximated? What can be solved in theory, but not in practice, and why not?
- **Simulation:** How does one use an automated or automatable model to predict or understand the behavior of some domain or system? Once abstracted how can one explore hypotheticals and generalizations of the original domain or problem space?
- **Automation:** What properties allow some kinds of models to be manipulated by a program and to be automatically executed? How do these special models connect us to certain real domains?

For core computationalists for whom the typical computing curriculum centers on the machine, we propose that courses also include a focus on models and languages—the intellectual frameworks of computationalist thinking. For the contextualized computationalists, curricula grounded in principles of computationalist thinking tailored to domain-specific needs has the potential to be transformative, not only by encouraging innovation within a domain but also by creating entirely new disciplines. Lastly, at both the secondary and post-secondary levels, we urge that a minimal level of computationalist literacy be required of all students.

7.2 What Everyone Should Know

Learning outcomes for every student should include an exposure to the ideas of modeling, abstraction, and automation as discussed above. The exposure provides insight into the tight relationship between models, languages and machines. Students should be exposed to different levels of abstraction and representation, understanding how to create symbolic, graphical or numerical representations of relationships and data; and understand how this can be used to clarify comprehension of complex sets of information.

Every student should understand enough of the issues of scales and limits to appreciate that some problems are easier than others and some notion that there are good systematic reasons for this. We expect that notion of computational models are stand-ins for real processes will expose the student to the idea of simulation and help her appreciate that simulation is occurring in common every day uses of their computing devices.

In addition, there is a wide range of practical skills that involve computing. Although we support as laudable educating students so as to demystify their computing devices, to expose them to ethical implications of using such devices, and to allow them to best use computers as tools, a thorough discussion is beyond the scope of this document.

7.3 What Contextualized Computationalists Should Know

Practitioners in other fields often build expressive and descriptive models of physical, human, or abstract systems. Contextualized computationalists build and compute with those models. Sociologists employ sophisticated graph-theoretic techniques for the

[5] Pronounced COM-pure-et (or kɑmpjʊrət in IPA), meaning having the ability to understand computationalist thinking and having facility with computing, as literate and numerate are for reading/writing and numbers, respectively.

modeling and analysis of social networks; however, a computational sociologist is able to develop and apply these models to larger scale problems allowing for more complex link analysis. Systems biologists discover the emergent properties from the complex interactions of biological systems. A computational biologist can take a population's DNA microarray data and mine that data to gain fundamental insights into the genetic and environmental causes of diseases. Economists model financial transactions at the micro and macro levels. A computational economist can model millions of individual agents in the economy to validate or refute macroeconomic theories.

Effective modeling requires that one be able to (1) state a problem clearly and precisely, (2) develop and understand a model, (3) compute with that model, and (4) understand and present the results. In step 1, one needs to determine what are the questions worth asking and which ones are practical to answer. Step 2 requires that one properly characterize inputs (*e.g.*, symbolic vs. numeric, discrete vs. continuous, understood and complete vs. uncertain with missing data, and so on) and consider the expected properties of the transformations over the model. In Step 3, a student needs to recognize and use a broad repertoire of approaches. Step 4 requires some consideration of the larger system that includes end users, including interface design, visualization, and so on.

Computing provides the methods and tools necessary to manage huge amounts of data, share this data with a global community, and use algorithmic approaches to extract meaning from this data. Given a problem, one needs to be able to describe the relationship between problem size and the resources necessary. Computing enables the processing of data at many different scales, but a contextualized practitioner must recognize the pragmatic and theoretic limits of computation.

Thus, the contextualized computationalist must understand in more depth models, scales and limits, and abstraction, particularly as it applies to her domain. For many contextualized computationalists, further facility in simulation is necessary, particularly for the purposes of prediction and exploring hypotheticals. She is more than a user of systems, however, because she must be able to extend and modify simulations.

Because the contextualized computationalist is still a non-computing professional, she needs to understand the limitations of her knowledge and skills and know when and how to approach a dedicated computationalist. For more fruitful collaboration, she must be able to communicate in the same language as the dedicated computationalist as well as in her own domain speciality.

Finally, a contextualized computationalist must understand how to use a computing device responsibly and consider the implications of data misuse. She should appreciate the impact of computing in enabling the products she uses everyday.

7.4 What Computationalists Should Know

We emphasize that computing is a broad field. There are several specialized sub-disciplines within it. Here, we seek only to outline a core set of knowledge that most computationalists should share in order to understand the field and to work effectively within their chosen discipline.

The starting point for this shared understanding is the comprehension of themselves as computational modelers who use abstract languages to transform states and processes through a computational machine. It is not clear that we currently emphasize this view in any of our typical programs, and we should.

There is a large body of knowledge that is beyond the scope of this document to enumerate further than we have above, but must be understood by computationalists. Each sub-discipline will require differing levels of depth in these areas and will also have their own specific additional topics. As modeling is a key characteristic of computationalists they must understand the mathematical foundations of computational modeling and how to create, analyze and critique models.

Computationalists do not need a complete understanding of hardware architecture any more they need to have a complete understanding of larger computing systems that take into account humans as well as their computational devices; however, they must have a clear understanding of how computing devices work as an abstract machine. The level of abstraction and depth of knowledge will depend on their specialist area of computing. Computationalists must also understand the issues of distributed modeling—as in, for example, parallel and distributed processing, or networked systems—and the benefits and complexities these add to computational systems.

Computationalists create models to solve problems. Consequently they must understand the general principles of developing such models. In some disciplines this involves systems development processes and in others less so; regardless, computationalist must be able in varying degrees to:

- analyze a problem to understand the context and requirements;
- design a solution to that problem and implement that solution using appropriate tools and techniques;
- verify that the solution--which may or may not be a computer program--behaves correctly; validate that it meets its intended requirements;
- identify erroneous components and correct the problems in those components;
- document the solution's development to enable others to understand the rationale for decisions made during development; and
- manage the development process, including being able to make informed estimates on the difficulty of development.

Note that many of these *can* apply to computer programs but need not be understood solely in that context.

As professionals practicing in the field, computationalists must understand the issues of monitoring their processes and practices to ensure the quality of the result being produced. They must also understand the issues involved in system evolution in order to be able to make appropriate decisions about trade-offs. Designing and implementing a solution requires that computationalists know the performance constraints of their computing environment in order to make informed decisions about the feasibility of a solution or how to best structure it.

Of course, not all computationalists produce artifacts in the same way. For example, those focused on theoretical pursuits may do little in the way of system development and deployment. For

those computationalists, the meaning of managing processes may be different than described above, or emphasized in a different way. Therefore, the reader should not take these requirements to imply that computationalists must be able to apply a software engineering methodology or be able to use a traditional third-generation programming language.

Finally, it should be understood that a key characteristic of computing is that it enables those in other domains to be more effective and to solve problems of a scale previously impossible to consider; a computationalist's role is to facilitate others' activities, which requires good communication and negotiation skills and the ability to quickly gain insights into other domains. Insofar as the computationalist will solve problems in a domain outside of the field of computing, it is important that computationalists are aware of the impact that their activities will have on those domains and the ethical implications of their actions. This requires a general understanding of the evolution of computing and its impact on society.

7.5 Conveying What We Should Know

It is beyond the scope of this paper to provide a detailed map of the many ways in which the ideas described above might be embodied in curriculum. In this section, we briefly visit sample curricula that capture some aspects of this approach. These are neither the only ways in which computationalist thinking might be embedded in curriculum nor necessarily the best ways. We hope that they will help to make possible approaches concrete and to serve as food for thought as further possibilities are explored.

In Section 3, we noted a number of efforts at curricular reforms driven by the same needs that inspired our group. In particular, we identified Georgia Tech's Threads curriculum, Olin's "small footprint" curriculum for computing, several programs on digital media computation, and on applications of computing to science.

Olin's small footprint curriculum is built around a three-course core that refactors the material traditionally covered in program design, theoretical computer science (including algorithms and programming languages), and software systems. This refactoring underscores connections and themes such as those described in section 7.1. Students encounter each of the six key ideas described above in each of these classes. For example, in software design they select appropriate models out of which to construct programs; in foundations of computer science they describe tradeoffs among programming languages, data structures, and formal representations; and in software systems they encounter the different ways various models provide analyze and evaluate system properties.

The small footprint curriculum shifts focus from specific artifacts and technologies to the key ideas of computationalist thinking. It affords the opportunity to demonstrate connections among topics that are not always clear in a conventional curriculum. For example, trees—with their logarithmic/exponential structure—underlie phenomena as diverse as parsing, NP-completeness, searching and sorting, and declarative programming. At the same time, a small footprint curriculum forces decisions about what to omit. Olin's curriculum does not aim to teach students everything they might need to know ("just in case" learning); instead, it provides what they need to know in order to learn the rest ("just in time").

The Thread curriculum developed at Georgia Tech takes a different approach to the problem of computationalist thinking. Threads are partial paths through a computing degree. Each embodies a flexible set of technical skills both within and outside of computing that (1) serve as a context for interpreting the courses in a curriculum and (2) suggest a coordinated path through courses so that the end result is expertise in the area of the thread. Every student constructs her own personalized computing degree by weaving two threads. Each Thread is about 2/3 of a degree, but any pair of threads yields a complete degree. The Threads model represents extending the application of contextualization from courses to an entire undergraduate computing degree. Each thread defines its own set of courses and so provides an opportunity for each to define its own basic core. Thus, each thread can define a context for creating specific models consistent with the areas it touches. The Intelligence thread can concentrate on modeling intelligent behavior, the People thread on modeling the cognitive and physical capabilities of humans using computer systems, and so on.

In Threads, the curriculum designer seeks to avoid the problem of defining a core set of knowledge for all computationalists by allowing each thread to define its own while still requiring that every pair of threads is compatible. At Georgia Tech this has worked well. Although Georgia Tech's CS degree has no core, the intersection of threads essentially defines a small common set of beginning courses (a natural consequence of the 2/3+2/3=1 rule and the fact that each Thread is still about computing). From there the basic computationalist core of the sort we have described in 7.4 can be explored no matter what the combination of threads taken by the student.

Computational media and computational science curricula are two curricular families that occupy the space directly addressed in section 7.3: contextualized computation. Although focused on specific problems in media, science or engineering, such curricula must explicitly expose students to the computational aspects of the models in their domains of interest, emphasizing scales, limits, and abstraction. This exposure is sometimes done purely through the practice of programming, perhaps in the most popular languages of the field. To be truly effective, however, we would argue that the computational scientist (or computational media expert) must be able to explicitly connect those languages to abstract and executable models in their own domains. These connections need not be made explicit early in the curriculum, but must be made explicit eventually. For example, Georgia Tech's computational media degree looks remarkably like one of the CS threads (the Media thread) combined with a set of advanced courses drawn from the CS degree and from the School of Literature, Communication and Culture. Thus, such students have exposure to the same core and ideas as CS majors but this core is explicitly framed in terms of media, models of media processes, and representations that capture those processes.

These approaches are hardly exhaustive of the ways in which computationalist thinking might inform a curriculum. Exploring this broader space—explicitly building from the principles we outline here, starting anew to define the curricula that result—is a next step in the task of (re)defining computing curricula.

8. Where Should We Go From Here?

Adrion et al. suggest [3] that:

> In identifying the bare essentials and enduring funda-

mentals of any discipline, one needs to understand (i) the kinds of questions it cares about, (ii) the kinds of results it accepts as answers, (iii) the methods it prefers to seek the results, and (iv) the kinds of evidence it accepts to validate the results.

This paper has begun to address the questions that computationalists ask and the ways that we approach them. Our efforts are, of course, only a beginning. Specific curricula and courses need to be defined; explicit organizations of knowledge should be argued and proposed; and mechanisms for assessment must be developed.

This conversation must be continued among a wider group of stakeholders. These include not only computing educators, but also educators in other disciplines, computing professionals, secondary school teachers, professional organizations, and policy makers. While we recognize that there may be disagreement over particular terminology, starting from the idea that being computationally literate is one of the hallmarks of an educated society will help to bridge differences in semantics.

At a minimum, the curriculum of existing courses should be revisited to inculcate computationalist thinking—specifcally, core competencies in modeling, scales and limits, simulation, abstraction, and automation. To truly embrace the focus on models, languages, and machines as a single computationalist idea, we will in most cases need to radically rethink curricula to better reflect this way of organizing and articulating the topics in our field.

One exercise that this working group undertook was to examine ACM's CC2005 recommendation [15]. There is no doubt, as in any other field, that the definition of the core knowledge varies among educators, education programs, institutes, and regions; nevertheless, CC2005 provides one standard reference for structuring (and debating) computing curricula. Our group examined CC2005's suggested set of distinguished computing areas for all (see Table 3.2 in CC2005), and an additional set for computing professionals (see Table 3.1 in CC2005). In both cases, we found the tables significantly lacking in topics that are crucial to our vision of computing.[6]

The exercise proved fruitful (and unsurprisingly, not entirely uncontroversial); however, the discussion revealed a larger issue. Given our emphasis on modeling, it would make sense that topics involving modeling, humans, and so on should be reflected in such a table. More to point, our emphasis on modeling suggest a completely different organization of those areas. The current organization tends to be centered more about machines and engineering of those machines rather than around models and the development and applications of those models. We expect that reorganizing those in that way would lead to a completely different perspective and organization of even the more traditional computing topics.

Finally, we need to be able to communicate the importance of computationalist thinking to a wide range of audiences who may or may not be computing professionals. To this end, we have provided a first draft at a position paper aimed specifically at policy-makers (rather than educational specialists). We invite the creation of other similar documents—for example, to stakeholders in other non-computing domains—that will help influence and shape the discussion.

[6] Although we do not reproduce them here, our exercise resulted in over a hundred additional knowledge areas not covered by these two tables.

9. Acknowledgements

This effort is based in part upon work supported by the National Science Foundation under Grant Number IIS-0946665. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The authors would also like to acknowledge Michael Littman for engaging the group in early conversations on this topic and Mark Nelson for explaining IPA pronunciation and for the discussion that led to coining "compurate".

10. REFERENCES

[1] Abelson, H., and Sussman, G. J. 1985. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass.

[2] Adelman, L. 1994. Molecular computation of solutions to combinatorial problems. *Science* 266, 1021-1024.

[3] Adrion, R., Aiken, B., Bernat, A., Brown, J., Cooper, S., Dunn, M., Finlay, M., Giles, R., Gries,R., Kelemen, C., Krishnamurthy, S., Kumar, D., Kurose, J., Lawrence, A., Masi, L., McCracken, D., Merritt, S., Murtaugh, T., Plotkin, J., Prey, J., Ryder, B., Siraj, R., Stein, L., Tao, L., Teller, V., Thomas, J., Topi, H., Sutner, K., Shaw, M., and Wolz, U. 2006. Report of the NSF Workshop on Integrative Computing Education and Research (Northeast Workshop). Cambridge, Massachusetts, November 2005/January 2006.

[4] Bareiss, C., Powers, K., Thede, S., Meredith, M., Shannon, C., and Williams, J. 2004. The Computer Science Small Department Initiative (CS_SDI) Report. *SIGCSE Bull.* 36(1), 332-333.

[5] Bentley, J. 1986. Little Languages. *Communications of the ACM*, 29(8), 711-21.

[6] Brady, A., Bruce, K., Noonan, R., Tucker, A., and Walker, H. 2004. The 2003 model curriculum for a liberal arts degree in computer science: preliminary report. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, 282-283.

[7] ACM/IEEE-CS Joint Curriculum Task Force. Computing Curricula 2001: Computer Science Volume. 2001. *Journal on Educational Resources in Computing* 1(3).

[8] Denning, P. 2003. Great principles of Computing. *Communications of the ACM*, 46(11), 15-20.

[9] Downey, A., and Stein, L. 2006. A Small Footprint Curriculum for Computing. *Frontiers in Education*, San Diego, California, October 2006.

[10] Felleisen, M. , and Krishnamurthi, S. 2009. Why Computer Science Doesn't Matter. *Communications of the ACM* 52(7), 37-39.

[11] Foley, J. 2002. Computing > Computer Science. *Computing Research News* 14(4).

[12] Furst, M., Isbell, C., and Guzdial, M. 2007. Threads: How to Restructure a Computer Science Curriculum for a Flat World. In *Proceedings of the Thirty-Eighth Technical Symposium on Computer Science Education.*

[13] Guzdial, M. 2003. A Media Computation Course for Non-Majors. In *Proceedings of the 6th Annual Conference on Innovation, and Technology in Computer Science*, 104-108.

[14] NRC. 2004. Computer Science: Reflections on the Field, Reflections from the Field. Committee on the Fundamentals of Computer Science: Challenges and Opportunities, Computer Science and Telecommunications Board, National Research Council, National Academies Press 2004.

[15] Shackelford, R., McGettrick, A., Sloan, R., Topi, H., Davies, G., Kamali, R., Cross, J., Impagliazzo, J., LeBlanc, R., and Lunt, B. 2006. Computing Curricula 2005: The Overview Report. *SIGCSE Bull.* 38(1), 456-457.

[16] Stein, L. A. 1999. Challenging the Computational Metaphor: Implications for How We Think. *Cybernetics and Systems* 30(6), 473-507.

[17] Stein, L. A. 1999. What We Swept Under the Rug: Radically Rethinking CS1. *Computer Science Education*, 8(2), 118-129.

[18] Tucker, A. Deek, F., Jones, J., McCowan, D., Stephenson, C., and Verno. A. 2003. A Model Curriculum for K–12 Computer Science: Final Report of the ACM K–12 Task Force Curriculum Committee.

[19] Wing, Jeannette. 2006. Computational Thinking. *Communications of the ACM* 49(3), 33-35.

[20] Wolz, U., Domen, D., and McAuliffe, M. 1997. Multi-Media Integrated into CS 2: an Interactive Children's Story as a Unifying Class Project. *SIGCSE Bull.* 29(3), 103-110.

[21] Xu, Y, editor. 2009. Transform Science: Computational Education for Scientists. Microsoft Research 2009. DOI=http://research.microsoft.com/transformscience/CEfS.pdf

Appendix A

A Preliminary Policy Document Based on the Report

What is Computing?

Computing is fundamentally about creating and using models to simulate and explore actual or theoretical phenomena. Through the design, manipulation, and interpretation of these models, computing not only facilitates a better understanding of the processes and data represented, but is also transformative by its very nature. It provides a context for innovation in a wide range of other disciplines and often serves as the basis for the creation of entirely new fields.

Computing is interesting, engaging, and relevant to students. Continuous research and advancements in pedagogy and the applicability of computing to the real world provide a compelling and motivating environment for students to learn. Furthermore, the typical project-based teaching methodology used strengthens both higher-order thinking skills such as abstraction, critical thinking, and algorithmic problem-solving, as well as soft skills such as project organization, time management, teamwork, and collaboration. In all cases, computationalist thinking helps to create a more well-prepared student who will have increased success in either higher education or the workforce.

Computing is not the use of a computer. While students must certainly acquire the skills necessary to use the technology that surrounds them, students must also gain a fundamental knowledge and understanding of models and representations and a computationalist way of thinking about them.

Computing in Crisis

Yet despite a clear need for computationalist thinking, an exponential proliferation of computers, and a continually increasing reliance on technology, the study of computing is in crisis.

Although computing-centered occupations comprise three of the top six fastest growing occupations and provide above-average salaries, there is a significant lack of qualified candidates for these jobs due to long-term declining enrollments in computing programs and concerns about job outsourcing. Even in fields where a significant knowledge of computing is essential, a narrow belief that computing is only about the computer creates a misinformed perception among students that computing is both uninteresting and irrelevant.

In secondary education, the lack of core credit and an almost single-minded focus on programming discourages students from the study of computing. Since 2002, the number of students taking the Advanced Placement Computer Science exam decreased by 12.5% while the students taking AP Latin increased by 28.2%. In 2008, eighteen times as many students took AP Calculus and six times as many took AP Psychology as AP Computer Science. In 2005, the NCAA eliminated computer science as an acceptable course for determining initial eligibility of student-athletes. When they exist at all, computing classes are often relegated to a less academic business or vocational track.

At all levels, diversity and equity remain significant challenges. Participation rates of women and underrepresented minorities in computing are not only extremely low but have decreased more quickly than for the general population.

Three Distinct Audiences

The educational needs of students differ according to their field of study and level of schooling. Each audience requires an individual set of computationalist competencies in order to meet their specific needs.

The first audience consists of the core computationalists. These post-secondary students specialize either in traditional computing fields such as computer science, software engineering, computer engineering, information science, and information technology or in disciplines where computing is a central focus such as multimedia computation. Individuals in this group often work as theorists, researchers, practitioners, or developers of cutting-edge technologies in both academia and industry.

The second audience contains the contextualized computationalists. These are typically post-secondary students who require more in-depth knowledge and understanding of particular aspects of computing, but only as they apply to their particular domain. Examples include students of bioinformatics, computational economics, and technical management. In each case, the students' core field of study is not computing but rather a non-computing discipline strongly shaped and heavily influenced by computing principles.

The final audience includes the remaining post-secondary students and all secondary school students. A broad overview of computing knowledge is a fundamental component of being an educated person in the 21st century. The goal is not to instill technical proficiency, but to provide a basic computing context and an intellectual toolset for all students. Just as every educated student should understand basic scientific principles, have an historical perspective, be familiar with the basic canon of literature, and be able to communicate effectively, every student should also be able to think computationally.

A National Imperative

In order to adapt to the dynamic nature of technology and the rapid pace of technological change, it is essential that today's students have a broad understanding of computation, computational thinking, and algorithmic problem solving rather than be schooled in any particular technological skill set.

It is imperative that computationalist thinking be treated as a critical skill and knowledge set for students of the 21st century. Computing should be considered one of the new core disciplines on a level with reading, writing, math, and science. Education research funding should be focused on revising curricula and providing professional development for teachers at both the secondary and post-secondary levels to address the individual educational needs of each group of students. Cross-disciplinary interactions should be facilitated and curricular partnerships encouraged. Local and national leaders in education policy must set the direction and communicate the urgency of this imperative.