Support Collaboration by Teaching Fundamentals

Matthew Stone
Computer Science and Cognitive Science
Rutgers, The State University of New Jersey
110 Frelinghuysen Road, Piscataway NJ 08854-8019
Matthew.Stone@Rutgers.EDU

Abstract

This paper argues for teaching computer science to linguists through a general course at the introductory graduate level whose goal is to prepare students of all backgrounds for collaborative computational research, especially in the sciences. We describe our work over the past three years in creating a model course in the area, called *Computational Thinking*. What makes this course distinctive is its combined emphasis on the formulation and solution of computational problems, strategies for interdisciplinary communication, and critical thinking about computational explanations.

1 Introduction

The central long-term challenge of computational linguistics is *meaningfulness*. I want to build situated, embodied interactive agents that can work with people through language to achieve a shared understanding of the world. We have an increasing toolkit to approach such problems. Linguistics gives us powerful resources for representing utterance structure and interpretation, for example through the family of formalisms and models that have grown up around dynamic semantics and discourse representation theory. Supervised machine learning has proved to be a profoundly successful software engineering methodology for scaling representations and models from isolated instances to systematic wide coverage. Nevertheless, talking robots are a long way off. This is not a problem that is likely to be solved by writing down a corpus of interpretations for sentences (whatever that might mean) and training up the right kind of synchronous grammar. Nor is it likely to be solved by some one lone genius—half Aravind Joshi, half Richard Stallman—driven to learn and implement solo all of linguistics, artificial intelligence and cognitive science. Progress will come through teamwork, as groups from disparate backgrounds come together to share their discoveries, perspectives, and technical skills on concrete projects of mutual interest. In the course such collaborations, I expect research to unlock fundamental new insights about the nature of meaning, about its dependence on perception, action, linguistic knowledge and social relationships, and about the architecture of systems that can pick up on, create, and generalize meanings in their experience. This paper offers an interim summary of my reflections on preparing the next generation of scientists for this endeavor.

My efforts are anchored to the specific community where I work. Semantics at Rutgers involves a core group of eight faculty from linguistics, philosophy and computer science, with a committed group of about twice that many PhD students. That's three or four students a year: not much if you're thinking of running a class for them, but pretty big if the aim is to place graduates, as we successfully have recently, in positions where they can continue to do semantics (that is, in academic research and tenure-track faculty jobs). Interdisciplinary interaction is the norm for our group; it means that each semantics project inevitably introduces the team to questions, concepts and methodologies that lie outside the background expertise its members bring to the project as individuals. My own work is a good ex-

Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics (TeachCL-08), pages 129–136,
Columbus, Ohio, USA, June 2008.

ample: papers like (DeVault et al., 2006) or (Lepore and Stone, 2007) reveal interconnections between computational ideas and philosophical analysis that my colleagues and I discovered in attempting to bridge our different perspectives on meaning and meaningfulness.

In my experience, what makes it possible for these efforts to take up sophisticated computational ideas is not getting everyone up to speed with a specific programming environment or linguistic formalism. The key step is to get outsiders to appreciate the arguments that computer scientists make, and why they make them. Jeannette Wing (2006) calls this *Computational Thinking*. Wing argues that you should have a course where you teach first-year college students to think like computer scientists. But her arguments apply just as cogently to graduate students in the sciences, and to linguists in particular. Computation as a framework for data collection, data analysis, inference, and explanation has become the norm in the physical and life sciences, and is rapidly transforming the behavioral sciences and especially now the environmental sciences. The situation is not so different in the cultural fields of media, arts and entertainment either—as video game designers are quick to remind us. A wide swath of researchers in any university are now interested in supporting exploratory and innovative interdisciplinary computing research, and specifically in training future faculty to pursue and mentor such collaborations. We decided to make common cause with them at Rutgers, since computational linguistics is such a small group. So our computer science department offers a general course called *Computational Thinking* at the introductory graduate level, aimed at preparing researchers across fields to work on collaborative projects involving computational research. You have an opportunity to do the same.

2 Overview of the Course

We hold *Computational Thinking* in three hour blocks once a week. This responds to Rutgers's quirky geography, with philosophy, linguistics and computer science each on different campuses along a five-mile stretch of the main local thoroughfare, route 18. Elsewhere, it might make more sense to meet in more frequent, shorter sessions.

Each meeting is divided so that students spend about half of each lecture session (and half of each week's work) on technical material drawn from the standard computer science curriculum. As outlined in Section 2.1, the technical material mixes programming practice, problem-solving techniques and theoretical tools, and aims to provide the key elements that are needed to appreciate the computational considerations of an interdisciplinary research project. The typical format of these sessions is live interactive literate programming. We work in Scheme, supported by the DrScheme system available at http://www.plt-scheme.org/software/drscheme/. I beam an image of the Scheme development environment in front of the class and write, run, analyze and debug short programs on the fly. Students follow along on their laptops, doing exercises, asking questions, and seeing results as they go.

The remainder of each class meeting (and the associated outside coursework) explicitly focuses on the interpretive effort and people skills required to reframe the ideas and methodologies of another field in computational terms. Partly, as outlined in Section 2.2, that involves developing a shared understanding of how computers accomplish the representation, processing and problem solving they do, so that students become comfortable at viewing computational systems abstractly as manipulating generative scientific models and knowledge. Fundamentally this understanding is what enables an interdisciplinary team to reconcile the principles of an outside field with the practice of computer science. In addition, as outlined in Section 2.3, we offer explicit discussion of the conversational strategies and interactive skills involved in bridging the different perspectives of an interdisciplinary team, and overcoming the divides of disjoint academic cultures, the stresses of work and deadlines, and the many possibilities for misunderstanding.

Homework mixes short benchmark problems, which allow students to assess their progress against objective standards, with open-ended collaborative assignments that let students apply their domain expertise and challenge their developing skills and programming, problem solving and teamwork. This year students worked individually on a set of exercises on list processing, matching of recursive struc-

tures, and interpreting programming languages designed to give some general competence in Scheme. Then they worked in teams of three to four to develop a web site using DrScheme's Scheme servlet architecture. Finally, they explored the possibilities for computational research in their home field in a brief speculative paper.

The course has been offered twice, with about a dozen students participating each session. Three or four each year—the expected number—come from linguistics and the philosophy of language. The small numbers nevertheless add up. Already more than half the students in this spring's dissertation reading group in the philosophy of language had taken *Computational Thinking*. The group's focus was context, and the related problems of common ground, presupposition, anaphora and accommodation. You could feel the difference *Computational Thinking* made for many of the students, philosophers included, who succeeded not only in framing computational arguments about context and context change, but also in synthesizing computational concerns with philosophical ones in explaining linguistic interpretation in terms of context.

2.1 Technical ideas

The technical goal of the course is to give students greater facility in stating problems in computational terms and understanding and building solutions to computational problems. The perspective aligns with the online textbook *How to Design Programs* (Felleisen et al., 2001), which accompanies the Dr Scheme distribution, but we emphasize its continuity with the general mathematical problem solving that students have been doing since elementary school (Polya, 1945). Indeed, following Wing (2006), we see computational thinking as ordinary and pervasive. "It's not just the software and hardware artifacts we produce that will be physically present everywhere and touch our lives all the time, it will be the computational concepts we use to approach and solve problems, manage our daily lives, and communicate and interact with other people" (Wing, 2006, p. 35).

On our view, the main challenge of learning to think like a computer scientist—or to argue with one—is the abstraction and flexibility you need. For example, modern machine learning techniques amount to finding a solution to a problem that is partially specified in advance but partially determined by empirical evidence that is available to the system but not to the programmer. Thus we teach computational problem solving through case studies whose input and output gets progressively more and more abstract and remote from the programmer. The progression is suggested by the following examples, which we cover either by developing solutions in in-class literate programming demonstrations or by assigning them as programming projects.

- Answer a determinate mathematical question, but one whose size or complexity invites the use of an automatic tool in obtaining the results. The sieve of Eratosthenes is a representative case: list the prime numbers less than 100.
- Answer a mathematical question parameterized by an arbitrary and potentially open-ended input. Prototypical example: given a list of numbers determine its maximum element.
- Answer a mathematical question where the input needs to be understood as a generative, compositional representation. Given the abstract syntax of a formula of propositional logic as a recursive list structure and an interpretation assigning truth values to the atomic proposition letters, determine the truth value of the whole formula.
- Answer a question where the input needs to be understood as the specification of a computation, and thus fundamentally similar in kind to the solution. Write an interpreter for a simple programming language (a functional language, like a fragment of scheme; an imperative language involving action and state; or a logical language involving the construction of answer representations as in a production rule shell).
- Answer a mathematical question where the *output* may best be understood as the specification of a computation, depending on input programs or data. A familiar case is taking the derivative of an input function, represented as a Scheme list. A richer example that helps to suggest the optimization perspective of machine learning algorithms is Huffman coding. Given a sequence of input symbols, come up with programs that encode each symbol as a sequence of bits and decode bit sequences as symbol sequences in such a way that the encoded sequence

is as short as possible.

- Answer a question where *both input and output* need to be understood as generative compositional representations with a computational interpretation. Reinforcement learning epitomizes this case. Given training data of a set of histories of action in the world including traces of perceptual inputs, outputs selected and reward achieved, compute a policy—a suitable function from perception to action—that acts to maximize expected reward if the environment continues as patterned in the training data.

We go slowly, spending a couple weeks on each case, and treat each case as an opportunity to teach a range of important ideas. Students see several useful data structures, including association lists (needed for assignments of values to variables in logical formulas and program environments), queues (as an abstraction of data-dependent control in production rules for example), and heaps (part of the infrastructure for Huffman coding). They get an introduction to classic patterns for the design of functional programs, such as mapping a function over the elements of a list, traversing a tree, accumulating results, and writing helper functions. They get some basic theoretical tools for thinking about the results, such as machine models of computation, the notion of computability, and measures of asymptotic complexity. Finally, they see lots of different kinds of representations through which Scheme programs can encode knowledge about the world, including mathematical expressions, HTML pages, logical knowledge bases, probabilistic models and of course Scheme programs themselves.

The goal is to have enough examples that students get a sense that it's useful and powerful to think about computation in a more abstract way. Nevertheless, it's clear that the abstraction involved in these cases eventually becomes very difficult. There's no getting around this. When these students are working successfully on interdisciplinary teams, we don't want them struggling across disciplines to encode specific facts on a case-by-case basis. We want them to be working collaboratively to design tools that will let team members express themselves directly in computational terms and explore their own computational questions.

2.2 Interdisciplinary Readings

There is a rich literature in cognitive science which reflects on representation and computation as explanations of complex behavior. We read extensively from this literature throughout the course. Engaging with these primary sources helps students see how their empirical expertise connects with the mathematical principles that we're covering in our technical sessions. It energizes our discussions of knowledge, representation and algorithms with provocative examples of real-world processes and a dynamic understanding of the scientific questions involved in explaining these processes as computations.

For example, we read Newell and Simon's famous discussions of knowledge and problem solving in intelligent behavior (Newell and Simon, 1976; Newell, 1982). But Todd and Gigerenzer (2007) have much better examples of heuristic problem solving from real human behavior, and much better arguments about how computational thinking and empirical investigation must be combined together to understand the problems that intelligent agents have to solve in the real world. Indeed, students should expect to do science to find out what representations and computations the brain uses—that's why interdisciplinary teamwork is so important. We read Gallistel's survey (2008) to get a sense of the increasing behavioral evidence from a range of species for powerful and general computational mechanisms in cognition. But we also read Brooks (1990) and his critics, especially Kirsh (1991), as a reminder that the final explanations may be surprising.

We also spend a fair amount of time considering how representations might be implemented in intelligent hardware—whether that hardware takes the form of silicon, neurons, or even the hydraulic pipes, tinkertoys, dominoes and legos described by Hillis (1999). Hardware examples like Agre's network models of prioritized argumentation for problem solving and decision making (1997) demystify computation, and help to show why the knowledge level or symbol level is just an abstract, functional characterization of a system. Similarly, readings from connectionism such as (Hinton et al., 1986) dramatize the particular ways that network models of parallel representation and computation anticipate possible explanations in cognitive neuro-

science. However, we also explore arguments that symbolic representations, even in a finite brain, may not be best thought of as a prewired inventory of finite possibilities (Pylyshyn, 1984). Computational cognitive science like Hofstadter's (1979)—which emphasizes the creativity that inevitably accompanies compositional representations and general computational capacity—is particularly instructive. In emphasizing the paradoxes of self-reference and the generality of Turing machines, it tells a plausible but challenging story that's diametrically opposed to the "modular" Zeitgeist of domain-specific adaptive cognitive mechanisms.

2.3 Communication

Another tack to motivate course material and keep students engaged is to focus explicitly on interdisciplinary collaboration as a goal and challenge for work in the course. We read descriptions of more or less successful interdisciplinary projects, such as Simon's description of *Logic Theorist* (1996) and Cassell's account of interdisciplinary work on embodied conversational agents (2007). We try to find our own generalizations about what allowed these teams to work together as well as they did, and what we could do differently.

In tandem, we survey social science research about what allows diverse groups to succeed in bridging their perspectives and communicating effectively with one another. Our sourcebook is *Difficult Conversations* (Stone et al., 1999), a guidebook for conflict resolution developed by the Harvard Negotiation Project. It can be a bit awkward to teach such personal material in a technical class, but many students are fascinated to explore suggestions about interaction that work just as well for roommates and significant others as for interdisciplinary colleagues. Anyway, the practices of *Difficult Conversations* do fit with the broader themes of the class; they play out directly in the joint projects and collaborative discussions that students must undertake to complete the class.

I think it's crucial to take collaboration seriously. For many years, we offered a graduate computer science course on computational linguistics as a first interdisciplinary experience. We welcomed scientists from the linguistics, philosophy and library and information science departments, as well as engineers from the computer science and electrical and computer engineering departments, without expecting anyone to bring any special background to the course. Nevertheless, we encouraged both individualized projects and team projects, and worked to support interdisciplinary teams in particular.

We were unsatisfied with this model based on its results. We discovered that we hadn't empowered science students to contribute their expertise effectively to joint projects, nor had we primed computer science students to anticipate and welcome their contributions. So joint projects found computer scientists doing too much translating and not enough enabling for their linguist partners. Linguists felt like they weren't pulling their weight or engaging with the real issues in the field. Computer scientists grew frustrated with the distance of their work from specific practical problems.

Reading and reflecting on about generally-accessible examples goes a long way to bridge the divide. One case study that works well is the history of *Logic Theorist*—the first implemented software system in the history of AI, for building proofs in the propositional logic of Whitehead and Russell's *Principia Mathematica* (1910). In 1955–56, when Herb Simon, Allen Newell and Cliff Shaw wrote it, they were actually an interdisciplinary team. Simon was a social scientist trained at the University of Chicago, now a full professor of business, at what seemed like the peak of a distinguished career studying human decisions in the management of organizations. Newell and Shaw were whiz-kid hackers—Newell was a Carnegie Tech grad student interested in software; Shaw was RAND corporation staff and a builder of prototype research computers. Their work together is documented in two fun chapters of Simon's memoir *Models of My Life* (1996). The story shows how computational collaboration demands modest but real technical expertise and communication skills of all its practitioners. Reading the story early on helps students appreciate the goal of the computational thinking class from the beginning: to instill these key shared concepts, experiences, attitudes and practices, and thereby to scaffold interdisciplinary technical research.

To work together, Simon, Newell and Shaw needed to share a fairly specific understanding of the concept of a *representation* (Newell and Si-

mon, 1976). Their work together consisted of taking knowledge about their domain and regimenting it into formal structures and manipulations that they could actually go on to implement. The framework they developed for conceptualizing this process rested on representations as symbolic structures: formal objects which they could understand as invested with meaning and encoding knowledge, but which they could also realize in computer systems and use to define concrete computational operations. In effect, then, the concept of representation defined their project together, and they all had to master it.

Simon, Newell and Shaw also needed a shared understanding of the computational methodology that would integrate their different contributions into the final program. Their work centered around the development of a high-level programming language that allowed Simon, Newell and Shaw to coordinate their efforts together in a particularly transparent way. Simon worked *in* the programming language, using its abstract resources to specify formulas and rules of inference in intuitive but precise terms; on his own, he could think through the effects of these programs. Newell and Shaw worked to *build* the programming language, by developing the underlying machinery to realize the abstract computations that Simon was working with. The programming language was a *product* of their effort together; its features were negotiated based on Simon's evolving conceptual understanding of heuristic proof search and Newell and Shaw's engagement with the practical demands of implementation. The language is in effect a fulcrum where both domain expertise and computational constraints exercise their leverage on one another. This perspective on language design comes as a surprise both to scientists, who are used to thinking of programming paradigms as remote and arcane, and to computer scientists, who are used to thinking of them solely in terms of their software engineering patterns, but it remains extremely powerful. To make it work, everyone involved in the research has to understand how their judicious collaborative exploration of new techniques for specification and programming can knit their work together.

In the course of developing their language, Simon, Newell and Shaw also came to share a set of principles for discussing the computational feasibility of alternative design decisions. Proof, like most useful computational processes, is most naturally characterized as a search problem. Inevitably, this meant that the development of *Logic Theorist* ran up against the possibility of combinatorial explosions and the need for heuristics and approximations to overcome them. The solutions Simon, Newell and Shaw developed reflected the team's combined insight in constructing representations for proof search that made the right information explicit and afforded the right symbolic manipulations. Many in the class, especially computer scientists, will have seen such ideas in introductory AI courses, so it's challenging and exciting for them to engage with Simon's presentation of these ideas in their original interdisciplinary context as new, computational principles governing psychological explanations.

Finally—and crucially—this joint effort reflected the team's social engagement with each other, not just their intellectual relationships. In their decades of work together, Simon and Newell cultivated and perfected a specific set of practices for engaging and supporting each other in collaborative work. Simon particularly emphasizes their practice of open discussion. Their talk didn't always aim directly at problem-solving or design. In the first instance, the two just worked towards understanding—distilling potential insights into mutually-satisfying formulations. They put forward vague and speculative ideas, and engaged with them constructively, not critically.

Simon's memoirs also bring out the respect the teammates had for each others' expertise and work styles, especially when different—as Newell's brash, hands-on, late-night scheming was for Simon—and the shared commitment they brought to making their work together fun. Their good relationship as people may have been just as important to their success at interdisciplinary research as the shared interests, ideas and intellectual techniques they developed together.

These kinds of case studies allow students to make sense of the goals and methods of the course in advance of the technical and interpretive details. Not much has changed since *Logic Theorist*. Effective computational teamwork still involves developing a conceptual toolbox that allows all participants on the project to formulate precise representations and engage with those representations in computa-

tional terms. And it still requires a more nuanced approach to communication, interaction and collaboration than more homogeneous efforts—one focused not just on solving problems and getting work done but on fostering teammates' learning and communication, by addressing phenomena from multiple perspectives, building shared vocabulary, and finding shared values and satisfaction. These skills are abstract and largely domain independent. The class allows students to explore them.

3 Interim Assessment

The resources for creating our *Computational Thinking* class came from the award of a training grant designed to crossfertilize vision research between psychology and computer science. The course has now become a general resource for our cognitive science community. It attracts psychologists from across the cognitive areas, linguists, philosophers, and information scientists. We also make sure that there is a critical mass of computer scientists to afford everyone meaningful collaborative experiences across disciplines. For example, participation is required for training grant participants from computer science, and other interdisciplinary projects invite their computer science students to build community.

One sign of the success of the course is that students take responsibility for shaping the course material to facilitate their own joint projects. Our initial version of the course emphasized the technical ideas and programming techniques described in Section 2.1. Students asked for more opportunities for collaboration; we added it right away in year one. Students also asked for more reading and discussion to get a sense of what computation brings to interdisciplinary research, and what it requires of it. We added that in year two, providing much of the materials now summarized in Sections 2.2 and 2.3. In general, we found concrete and creative discussions aimed at an interdisciplinary audience more helpful than the general philosophical statements that computer scientists offer of the significance of computation as a methodology. We will continue to broaden the reading list with down-to-earth materials covering rich examples.

From student feedback with the second running of the class, the course could go further to get students learning from each other and working together early on. We plan to respond by giving an initial pretest to get a sense of the skills students bring to the class and pair people with partners of differing skills for an initial project. As always this project will provide a setting where all students acquire a core proficiency in thinking precisely about processes and representations. But by connecting more experienced programmers with novices from the beginning, we hope to allow students to ramp up quickly into hands-on exploration of specification, program design and collaborative computational research. Possible initial projects include writing a production rule shell and using it to encode knowledge in an application of identifying visual objects, recognizing language structure, diagnosing causes for observed phenomena or planning goal-directed activity; or writing an interpreter to evaluate mathematical expressions and visualize the shapes of mathematical objects or probabilistic state spaces.

Anecdotally, we can point to a number of cases where *Computational Thinking* has empowered students to leverage computational methods in their own research. Students have written programs to model experimental manipulations, analyze data, or work through the consequences of a theory, where otherwise they would have counted on pencil-and-paper inference or an off-the-shelf tool. However, as yet, we have only a preliminary sense of how well the course is doing at its goal of promoting computational research and collaboration in the cognitive science community here. Next year we will get our first detailed assessment, however, with the first offering of a new follow-on course called "Interdisciplinary Methods in Perceptual Science". This course explicitly requires students to team up in extended projects that combine psychological and computational methods for visual interaction. We will be watching students' experience in the new class closely to see whether our curriculum supports them in developing the concepts, experiences, attitudes and practices they need to work together.

4 Conclusion

Teamwork in computational linguistics often starts by endowing machine learning methods with mod-

els or features informed by the principles and results of linguistic theory. Teams can also work together to formalize linguistic knowledge and interpretation for applications, through grammar development and corpus annotation, in ways that fit into larger system-building efforts. More generally, we need to bridge the science of conversation and software architecture to program interactive systems that exhibit more natural linguistic behavior. And we can even bring computation and linguistics together outside of system building: pursuing computational theories as an integral part of the explanation of human linguistic knowledge and behavior.

To work on such teams, researchers do have to master a range of specific intellectual connections. But they need the fundamentals first. They have to appreciate the exploratory nature of interdisciplinary research, and understand how such work can be fostered by sharing representational insight, designing new high-level languages and thinking critically about computation. *Computational Thinking* is our attempt to teach the fundamentals directly.

You should be find it easy to make a case for this course at your institution. In these days of declining enrollments and interdisciplinary fervor, most departments will welcome a serious effort to cultivate the place of CS as a bridge discipline for research projects across the university. *Computational Thinking* is a means to get more students taking our classes and drawing on our concepts and discoveries to work more effectively with us! As the course stabilizes, we plan to reach out to other departments with ongoing computational collaborations, especially economics and the life and environmental sciences departments. You could design the course from the start for the full spectrum of computational collaborations already underway at your university.

Acknowledgments

Supported by IGERT 0549115. Thanks to the students in 198:503 and reviewers for the workshop.

References

Philip E. Agre. 1997. *Computation and Human Experience*. Cambridge.

Rodney A. Brooks. 1990. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15.

Justine Cassell. 2007. Body language: Lessons from the near-human. In J. Riskin, editor, *Genesis Redux: Essays in the History and Philosophy of Artificial Intelligence*, pages 346–374. Chicago.

David DeVault, Iris Oved, and Matthew Stone. 2006. Societal grounding is essential to meaningful language use. In *Proceedings of AAAI*, pages 747–754.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs: An Introduction to Computing and Programming*. MIT.

C. R. Gallistel. 2008. Learning and representation. In John H. Byrne, editor, *Learning and Memory: A Comprehensive Reference*. Elsevier.

W. Daniel Hillis. 1999. *The Pattern on the Stone*. Basic Books.

Geoffrey E. Hinton, David E. Rumelhart, and James L. McClelland. 1986. Distributed representations. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 77–109. MIT.

Douglas Hofstadter. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books.

David Kirsh. 1991. Today the earwig, tomorrow man? *Artificial Intelligence*, pages 161–184.

Ernest Lepore and Matthew Stone. 2007. Logic and semantic analysis. In Dale Jacquette, editor, *Handbook of the Philosophy of Logic*, pages 173–204. Elsevier.

Allen Newell and Herbert A. Simon. 1976. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126.

Allen Newell. 1982. The knowledge level. *Artificial Intelligence*, 18:87–127.

G. Polya. 1945. *How to Solve it*. Princeton.

Zenon Pylyshyn. 1984. *Computation and Cognition: Toward a Foundation for Cognitive Science*. MIT.

Herbert A. Simon, 1996. *Models of My Life*, chapter Roots of Artificial Intelligence and Artificial Intelligence Achieved, pages 189–214. MIT.

Douglas Stone, Bruce Patton, and Sheila Heen. 1999. *Difficult Conversations: How to Discuss What Matters Most*. Penguin.

Peter M. Todd and Gerd Gigerenzer. 2007. Environments that make us smart: Ecological rationality. *Current Directions in Psych. Science*, 16(3):170–174.

Alfred North Whitehead and Bertrand Russell. 1910. *Principia Mathematica Volume 1*. Cambridge.

Jeannette M. Wing. 2006. Computational thinking. *Communications of the ACM*, 49(3):33–35.