An Introductory CS Course for Cognitive Science Students

James B. Marshall
Computer Science Program
Pomona College
Claremont, California 91711
marshall@cs.pomona.edu

Abstract

This paper describes an introductory computer science course recently developed at Pomona College that is designed specifically for liberal arts students majoring in cognitive science with no previous technical background in programming. A key component of the course is the use of low-cost robots to illustrate ideas about the computational foundations of cognition in a hands-on way. We describe the pedagogical objectives of the course and outline the major topics covered, including the robotics section and its relationship to the other course topics.

Introduction

Pomona is one of a small but growing number of undergraduate liberal arts colleges to offer both a major and minor in cognitive science. The Department of Linguistics and Cognitive Science, which administers the major, comprises a core group of faculty with backgrounds in linguistics, philosophy, and psychology. Faculty from several other departments are affiliated with the program as well, including computer science, neuroscience, anthropology, and music. Students majoring in cognitive science are required to take a wide range of courses in linguistics, psychology, and philosophy, reflecting the inherently interdisciplinary nature of the field. However, no computer science courses of any type are presently required for the major, although students may count certain CS courses such as Artificial Intelligence, Neural Networks, or Theory of Computation as elective credit toward the major if they so choose. Unfortunately, these CS courses are designed primarily for upper-level CS majors, and assume a significant amount of programming experience or mathematical sophistication. Their prerequisites usually preclude interested cognitive science students from taking them.

The unavailability of a suitable CS course for cognitive science majors and the consequent lack of any CS requirement has until recently represented a serious deficiency of the program. This is especially ironic given the importance placed on the notion of *computation* as an organizing principle in modern cognitive science's conception of the mind. One could argue that the fundamental difference between cognitive science and other closely related fields such as cognitive psychology is the former's emphasis on viewing human cognitive processes as a type of computation, and on building explicit models of these processes in the form of computer programs. Furthermore, computation plays an important role in the philosophy of mind, with philosophical arguments and debates about the mind-body problem often assuming a firm grasp of Turing machines and their properties. Students graduating with a major in cognitive science need to have more than just a passing acquaintance with these ideas, and ought to understand, on a basic level gained through firsthand experience, what it means to design, implement, and evaluate computational models of cognition.

With these considerations in mind, we undertook to develop a new introductory computer science course that would address these needs. The course would be targeted at students with backgrounds in linguistics, psychology, philosophy, or neuroscience who intended to major or minor in cognitive science. Our belief was that for students to really understand what computation is all about, they need to get their hands dirty and learn to program. No previous programming experience would be presumed, but the goal would be to get students up to speed sufficiently quickly to allow them to write and experiment with programs that model various aspects of cognition, such as learning, memory, perception, and language. This course, entitled *CS 30: Computation and Cognition*, was offered for the first time in the Fall of 2002.[1]

[1]The course was originally called *Programming Methods in Cognitive Science*, but this title was deemed too narrow and was subsequently changed.

Course Objectives

Due to the relatively large number of courses that cognitive science students must take to fulfill their major and distribution requirements, we knew that CS 30 would be the only computer science course that most of them would be able to take, so we wanted to make the course as comprehensive and self-contained as possible. Given the non-technical backgrounds of most of the students, however, it was also important to maintain a realistic set of expectations as to the level of programming sophistication that could be reached within a single semester. Above all, we wanted to motivate the programming topics by concepts and examples arising from cognitive science and artificial intelligence. Some of the "big" questions that we wanted students to come to grips with are listed below:

- What is computation, and why is it important to cognitive science?
- What are *top-down symbolic* models and *bottom-up emergent* models of cognition, and how do they differ from one another?
- Are there any limitations to what can be computed, in principle?
- What might these limits imply about human or machine intelligence?
- What is *embodied* cognitive science?

Scheme, a modern dialect of Lisp, was chosen as the main programming language for the course, in part because symbolic approaches to modeling intelligence have traditionally played a central role in AI throughout its history. Furthermore, the simplicity of Scheme's syntax, its absence of strict type checking, and the ease of using an interpreted language make it a good choice for teaching programming concepts to beginners. After students have acquired some hands-on experience with symbolic programming in Scheme, they are in a much better position to understand many of the arguments that lie at the heart of the AI debate, such as Newell and Simon's physical symbol system hypothesis (Newell & Simon 1976) and Searle's Chinese Room thought experiment (Searle 1980), both of which revolve around the key notions of symbol and meaning. The use of Scheme also facilitates the presentation of Turing machines and computability, another important course topic.

Synopsis

The format of the course consisted of two 75-minute lectures and one 75-minute closed lab session per week. All lectures and labs were held in a computer classroom, making it possible to assign in-class programming exercises in order to provide immediate reinforcement of newly-introduced concepts.

A rough breakdown of course topics by week is given below:

- Basic Symbol Manipulation and List Processing (1 week)
- Recursion and Data Abstraction (4 weeks)
- The Turing Test (1 week)
- Nautral Language Processing (1 week)
- Turing Machines and Computability (1 week)
- Physical Symbol Systems and the Chinese Room (1 week)
- Object-Oriented Programming (2 weeks)
- Neural Network Models of Memory and Learning (1 week)
- Robotics and Embodied Cognitive Science (1 week)
- Braitenberg Vehicles (1 week)
- Subsumption Architecture (1 week)

The first five weeks of the semester focused on teaching students the fundamentals of programming, with a strong emphasis placed on recursion and data abstraction. Aside from its emphasis on functional programming, this material was not that different from what one might find in a typical introductory course for CS majors. For this part of the course, we relied mainly on the first half of the textbook *Concrete Abstractions* (Hailperin, Kaiser, & Knight 1999). During the sixth week, however, we began discussing topics more directly relevant to cognitive science, starting with the Turing Test. Students read Turing's classic article, *Computing Machinery and Intelligence* (Turing 1950), as well as Douglas Hofstadter's piece *A Coffeehouse Conversation on the Turing Test* (Hofstadter 1981a), and wrote a summary of their own reactions. This discussion in turn provided the background for an extended programming assignment that led students through the development of a small rule-based natural language processing system (Hailperin, Kaiser, & Knight 1999, section 7.6). This program, though relatively simple in structure, conveyed the flavor of many symbolic AI models in a concrete way—in addition to being a fun programming assignment in its own right.

We turned our attention next to Turing machines and computability, which students had already gotten a preview of from reading Turing's article. A chapter from John Casti's book *Searching for Certainty* (Casti 1991) served as a good accompanying reading assignment for this segment of the course. After discussing the Halting Problem, Church's Thesis, and the universality of computation, we developed a Turing machine simulator in Scheme, and used it to investigate the properties of "busy beaver" Turing machines. This in itself was

a nice illustration of the idea of universal computation, showing how one type of "machine" (*i.e.*, our Scheme program) could fully mimic the behavior of other machines (*i.e.*, Turing machines) when given appropriately encoded descriptions of them (*i.e.*, lists of symbols representing state transition rules).

In one of the lab exercises, students experimented with several five-state busy beaver machines, some of which run for tens of millions of steps before halting. The general lesson here is that very simple rules can give rise to surprisingly complex behavior—an extremely important idea that underlies many bottom-up emergent models of cognition. The flip side of this is that behavior that appears to be complex or purposeful does not necessarily imply the existence of complex underlying mechanisms. This idea has important implications for intelligence, and resurfaces again later in the course in the context of robotics and Braitenberg vehicles.

At this point, students were prepared to read and discuss two of the most famous papers in AI: Newell and Simon's *Computer Science as Empirical Inquiry*, in which they put forth their Physical Symbol System Hypothesis (Newell & Simon 1976), and Searle's *Minds, Brains, and Programs*, in which he introduces his Chinese Room argument against the claims of strong AI (Searle 1980). Just for fun, we also read Hofstadter's piece *A Conversation with Einstein's Brain* (Hofstadter 1981b), which adds some useful perspective to the debate.

The remainder of the course focused on the idea of emergence and bottom-up approaches to modeling intelligence. Approximately two weeks were devoted to object-oriented programming, which was motivated by the development in Scheme of some simple neural network models of memory and learning. One programming assignment involved implementing a simulation of McClelland's Jets and Sharks interactive activation memory model (McClelland 1981) and experimentally investigating its properties. Students read *The Appeal of Parallel Distributed Processing* by McClelland, Rumelhart, and Hinton (McClelland, Rumelhart, & Hinton 1986) as background preparation for this assignment. We also briefly covered pattern associator networks and the delta rule learning algorithm, although unfortunately time constraints prevented us from going any further into learning. Had more time been available, coverage of multi-layer neural networks and the backpropagation learning algorithm, reinforced by a suitable programming assignment, would have been appropriate at this point.

The final three weeks were devoted to the general topic of embodied cognitive science. A key objective here was to get the students to appreciate from firsthand experience the following very important insight: intelligent behavior depends critically on the *interaction* of a physical system with its environment (Pfeifer & Scheier 1999). To explore this idea, students were provided with Handyboard/LEGO robot kits, along with step-by-step assembly instructions taken from Fred Martin's textbook *Robotic Explorations* (Martin 2000). Using Handyboards, however, made it necessary for students to learn a new programming language—Interactive C—which differs from Scheme in several fundamental ways. On the one hand, exposing students to a significantly different computational formalism was a good thing, for this reinforced the view of computation as being independent of any particular computer language (although coverage of Turing machines accomplishes this just as well). On the other hand, shifting gears so late in the semester meant that there was not enough time for students to really absorb the new language. Luckily, this turned out not to be a serious hurdle for most of the students, but it nevertheless detracted somewhat from the continuity of the course topics.

In any case, our study of embodied cognition centered around two topics: Braitenberg vehicles and Brooks' subsumption architecture. We began by reading and discussing the first few chapters of Braitenberg's seminal book *Vehicles: Experiments in Synthetic Psychology* (Braitenberg 1984). Lab sessions and homework assignments were devoted to implementing some of the simpler types of creatures described in the book using light sensors on the Handyboard robots. During the first lab session, students built their LEGO robot chassis and familiarized themselves with the technical details of the Handyboard controller and its Interactive C software interface.

In the following week's lab, students started with a control program for very simple locomotion (*e.g.*, going straight at a constant speed), and then modified it to produce more interesting behaviors, such as having the robot wander in different directions and at different speeds depending on its light sensor readings, or using its bump sensors to avoid obstacles. This lab emphasized experimentation, and was therefore less structured than earlier labs. In fact, students quickly realized that experimentation was essential in trying to program their robots to exhibit "intelligent" behavior, such as seeking out a light source while not getting stuck, for it was simply too hard to predict in advance with any reliability how well a particular program would work—even one consisting of only a few lines of code. The overall behavior of the robot might be profoundly influenced by small modifications to its program, or by variations in its surrounding environment. For instance, the sample program shown below, taken from a follow-up homework assignment, results in surprisingly effective light-seeking behavior. Simply interchanging the variables

`right` and `left` in the last two lines, however, results in light-avoiding behavior.

```
/* motor and light sensor ports */
int LEFT_MOTOR = 1;
int RIGHT_MOTOR = 3;
int LEFT_EYE = 3;
int RIGHT_EYE = 4;

void main() {

  while (1) {

    /* read light intensity */
    int left = normalize(analog(LEFT_EYE));
    int right = normalize(analog(RIGHT_EYE));

    /* activate motors */
    motor(LEFT_MOTOR, right);
    motor(RIGHT_MOTOR, left);

  }
}
```

In addition to Braitenberg's work, students read the paper *Intelligence Without Representation* by Brooks, which describes the subsumption architecture and the behavior-based approach to robotics from a somewhat philosophical point of view (Brooks 1991). Students experimented with different subsumption-style control programs for their Handyboards during their last lab session and on a subsequent homework assignment. Many of the robot exercises assigned in CS 30 were based on labs and assignments originally developed by Meeden and Kumar (Kumar & Meeden 1998; Meeden 2002).

Conclusion

Having students build and program physical robots, even those as simple as the Handyboards, reinforced in a concrete way many of the ideas encountered throughout the course. For example, Braitenberg in his book emphasizes what he calls "the law of uphill analysis and downhill invention", meaning that synthesizing complex behaviors from simple mechanisms is often much easier than attempting to infer the nature of the mechanisms underlying some observed behavior. Students encountered this same phenomenon when experimenting with robots, busy beaver Turing machines, and also in the context of neural networks, where simple interconnected processing units collectively gave rise to quite complicated and surprising observed behaviors.

Although the use of robots is not essential to the success of a course like CS 30, our experience has convinced us that they can significantly enhance its effectiveness, especially as a vehicle for teaching concepts relating to emergence and bottom-up models of intelligence. However, we believe they could also be used just as effectively to illustrate aspects of top-down symbolic models. For example, we can easily imagine having students implement a production-system or logical-inference based controller for a robot, although we have not yet developed such an exercise ourselves.

All of the labs and homework assignments used in the Fall 2002 offering of CS 30 are available on-line at http://www.cs.pomona.edu/~marshall/cs30. We encourage others to use and adapt these materials for their own courses as they see fit, and would be especially interested to see the development of similar courses for cognitive science students at other institutions.

References

Braitenberg, V. 1984. *Vehicles: Experiments in Synthetic Psychology*. Cambridge, MA: MIT Press.

Brooks, R. A. 1991. Intelligence without representation. *Artificial Intelligence* 47:139–160.

Casti, J. L. 1991. Proof or consequences. In *Searching for Certainty: What Scientists Can Know About the Future*. William Morrow. Chapter 6.

Hailperin, M.; Kaiser, B.; and Knight, K. 1999. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. Brooks/Cole Publishing.

Hofstadter, D. R. 1981a. A coffeehouse conversation on the Turing Test. *Scientific American*.

Hofstadter, D. R. 1981b. A conversation with Einstein's brain. In *The Mind's I: Fantasies and Reflections on Self and Soul*. New York: Basic Books. Chapter 26, 430–457.

Kumar, D., and Meeden, L. 1998. A robot laboratory for teaching artificial intelligence. In Joyce, D., ed., *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*. ACM Press. http://mainline.brynmawr.edu/Robots/ResourceKit.

Martin, F. G. 2000. *Robotic Explorations: A Hands-On Introduction to Engineering*. Prentice-Hall.

McClelland, J. L.; Rumelhart, D. E.; and Hinton, G. E. 1986. The appeal of parallel distributed processing. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1*. Cambridge, MA: MIT Press. Chapter 1, 3–44.

McClelland, J. L. 1981. Retrieving general and specific information from stored knowledge of specifics. In *Proceedings of the Third Annual Conference of the Cognitive Science Society*, 170–172.

Meeden, L. 2002. Computer Science 63 home page. http://www.cs.swarthmore.edu/~meeden/cs63/f02/cs63.html.

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

Pfeifer, R., and Scheier, C. 1999. *Understanding Intelligence*. Cambridge, MA: MIT Press.

Searle, J. R. 1980. Minds, brains, and programs. *Behavioral and Brain Sciences* 3:417–424.

Turing, A. M. 1950. Computing machinery and intelligence. *Mind* 59:433–460.