Is Programming Worthwhile?

Kurt Eiselt
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
eiselt@cc.gatech.edu

Abstract

The construction of computer programs that exhibit intelligent behavior is an essential part of teaching introductory artificial intelligence. To minimize the role of programming is to deny students an understanding of how to convert vague theories of intelligence into working artifacts. Yet to require students to tackle AI problems of sufficient complexity to be educationally valuable demands that we give these students more preparation in "AI programming" than is traditionally given them. This paper describes a course offered at Georgia Tech which provides that preparation.

Do Bears Live in the Woods?[1]

Despite what the title suggests, this paper is not entirely about whether programming in the context of teaching artificial intelligence (AI) is worthwhile. This paper is also about a programming course we now teach at Georgia Tech which is a prerequisite to the introductory undergraduate AI course. In this paper, I first argue that programming is not merely worthwhile, but that it is essential to the teaching of artificial intelligence. I then argue that to provide the introductory AI student with sufficient background to attempt interesting AI programming tasks, a separate course is necessary. Finally, I present one such course currently being offered at Georgia Tech.

Hint: The Answer is "Yes"

To gain some insight on the question of whether programming is worthwhile in AI education, it is useful to explore how some of our better known AI educators define what AI is. If you look at the textbooks they have contributed to the field, you'll notice that most of them qualify their definitions by saying that many different definitions of AI are possible. Nevertheless, none of them hesitate to offer definitions of their own. For example, in one of the earliest textbooks on artificial intelligence, James Slagle defines artificial intelligence in this way:

> Research scientists in Artificial Intelligence try to get machines to exhibit behavior that we call intelligent behavior when we observe it in human beings. Since the machine is almost always a computer, Artificial Intelligence is a branch of computer science. (Slagle, 1971)

After declaring that AI is not the study of computers (but before introducing definitions of AI in which programming takes a back seat, or no seat at all), Margaret Boden suggests:

> It would be somewhat more accurate to say that artificial intelligence is the study of computer programs. Indeed, many workers approach it from the context of programming science, being drawn primarily by the challenge it offers to programming techniques. And no one seriously concerned with it can avoid detailed reference to programs and programming. (Boden, 1977)

The Handbook of Artificial Intelligence tells us that AI...

> ...is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior---understanding language, learning, reasoning, solving problems, and so on. (Barr & Feigenbaum, 1981)

Still another textbook defines AI as...

> ...the study of mental faculties through the use of computational models. (Charniak & McDermott, 1985)

Compared to the previous definitions, there is not as much of an obvious connection between this definition and programming. However, the textbook from which this definition comes devotes the second chapter to LISP and illustrates many points with LISPish examples. The author of another widely-used textbook on AI offers a similar definition:

> [AI is the] study of the computations that make it possible to perceive, reason, and act. (Winston, 1992)

[1] Well, perhaps "live" isn't the word most commonly used in this context, but you get the idea, no?

The author of this definition puts some apparent distance between AI and programming, but this same author has also co-authored a LISP programming tutorial to be used in conjunction with his textbook. Does this author really believe that programming and AI are as distant as his definition implies? The reader is heartily encouraged to leap to his or her own conclusion.

What these definitions hold in common is that they either specifically mention programming or the behavior of computers, or they come from authors to whom programming is clearly very important. In other words, according to the authors of these introductory textbooks, understanding how to create programs that exhibit intelligent behavior is not merely useful, it is by definition integral to the understanding of artificial intelligence.

The Georgia Tech Perspective

At Georgia Tech, all undergraduate computer science majors are required to take the introductory artificial intelligence course. Over the past few years, the laboratory component of this course has evolved from one in which students take existing micro-programs and modify the code to one in which students build these micro-programs from scratch.

Undoubtedly there have been some negative effects of this change. The instructor and the teaching assistant must work harder, both when the students are working on the programs as well as when the programs must be graded. Furthermore, building the programs from scratch means that students may be exposed to fewer programs than when they were only required to modify existing programs. In addition, students complain more about the workload than before, since the laboratory component is now demonstrably more difficult. Finally, students now complain frequently and vociferously that the programming assignments are underspecified.

Despite these drawbacks, most of the AI faculty at Georgia Tech (although not necessarily all of us) agree that the advantages of having the students do a significant amount of programming far outweigh the disadvantages. One of the most difficult problems we face in AI is the problem of turning relatively vague and under-specified theories of intelligence into systems that actually do something which might be considered intelligent behavior. Thus, the students' complaints about underspecification are appropriate. But the only way that students can appreciate the vast difference between specifying a compiler and specifying a natural language understanding system, for example, is to try (and maybe even sometimes to fail) to construct intelligent programs.

This increased emphasis on programming requires that we provide the students with better preparation in the AI programming language of choice, which at Georgia Tech is Common LISP. Back when our students were required only to modify existing programs, it was sufficient that they had only two weeks of exposure to LISP in a freshman programming course. Now however, those two weeks of LISP instruction prove to be not nearly enough; the students tend to be frustrated by their lack of knowledge of the programming language, and end up spending too much of their time in the AI course learning LISP instead of learning AI concepts. As part of a major revision of our undergraduate computer science curriculum, we have introduced a new required course which we hope will help to solve this problem. This course, which is typically taken by students in their sophomore year, but may be taken as early as the freshman year, is called "knowledge representation and processing," and it introduces the student to four topics: (1) the functional programming paradigm, (2) the LISP programming language, (3) software design issues such as decomposition, modularity, abstraction, and programming for other people versus programming for the computer, and (4) basic knowledge representation and processing concepts (e.g., lists, trees, relational networks, dumb search, and heuristic search). Two companion courses introduce topics in object-oriented programming and concurrent programming, respectively. Together, these three courses comprise the prerequisites for our junior-year course on programming language concepts. So, while our "knowledge representation and processing" course prepares undergraduates (as well as some graduate students) for the introductory AI course, it also serves to introduce these students to one of the several different programming paradigms we want them to be exposed to prior to the programming language concepts course.

We have only been teaching this course for one year now, and most of the students who have taken this course have not yet taken the artificial intelligence course, so we cannot say much about how well prepared these students actually will be. But we can say that before they enter the AI course, most of these students have demonstrated a good understanding of topics such as state-space search and relational network representations, and they have attempted to implement small but interesting game-playing programs using minimax search. Furthermore, most of the students have enjoyed learning what is to them a most unusual programming paradigm, despite the occasional complaint that the course is not practical because they will "never get a job writing LISP programs." Not only do we expect that all these students will learn more from their AI course than their predecessors did, but that the AI instructors will be able to cover more material in the AI course than they have in past years.

Conclusion

As defined by the authors of the textbooks that most of us have used at one time or another for teaching artificial intelligence, the construction of computer programs that exhibit intelligent behavior is an essential part of AI itself. To minimize or completely eliminate programming in these introductory artificial intelligence courses is to deny students an understanding of how to convert vague theories of intelligence into working artifacts. Yet to require students to tackle AI problems of sufficient complexity to be educationally valuable demands that we give these students more preparation in "AI programming" than is traditionally given them. At Georgia Tech, we believe we have taken a significant step toward solving these problems by introducing a ten-week course which introduces the functional programming paradigm, the Common LISP programming language, basic software engineering

principles, as well as some of the fundamental concepts of AI programming.

The Course Syllabus

The "knowledge representation and processing" course was drawn in part from the classic text by Abelson and Sussman (1985), *Structure and Interpretation of Computer Programs.* In fact, it was the text we originally used for the course, but we have since dropped the text for two reasons: (1) the computer science students did not like the examples, which seemed better oriented toward engineering students, and (2) the book's examples were presented in the Scheme dialect while we asked the students to write their programs in Common LISP. Below is the syllabus for this course:

Course Description: CS 2360 is one of the new courses in the undergraduate computer science curriculum, and is intended for relatively new students to computer science. The goal of this course is not so much to help you learn to write efficient computer programs, but rather to help you learn to be efficient computer programmers. Another way of putting this is that you will learn to write programs for the benefit of people, as opposed to writing programs for the benefit of computers. As you will see during this course, these two ideals will often be in conflict. Along the way, you'll be introduced to a different paradigm for problem-solving and programming than the one you're probably most familiar with; this new paradigm is often referred to as functional programming, although we'll stray far away from the purist's approach to functional programming in this course. You'll also become intimately aware of program design principles of decomposition, modularity, transparency, and abstraction. In this sense, this course can be viewed as a software engineering course. At the same time, you'll also gain experience with diverse methods of representing information, particularly symbolic information, as well as different ways of manipulating that information, so you might also view this course as an algorithms and data structures course.

Our main vehicle for exploring this terrain will be the programming language called Common LISP. Common LISP is the second oldest computer programming language currently in widespread use, but don't let the age fool you--some implementations of Common LISP offer the most sophisticated programming environments in existence. LISP is usually closely identified with the world of artificial intelligence, as it is the language most often used in the development of "intelligent" systems. Yet LISP has also gained acceptance as a tool for the rapid prototyping of other large systems such as compilers and operating systems. In any case, knowledge of Common LISP will give you an additional tool for solving problems you may be faced with in your career.

Week 1

Tues
- Introduction to the course
  - motivation--why this course exists
  - why LISP?
  - functional programming v. procedural programming
  - interpreters v. compilers
  - programming for people v. programming for computers
  - course requirements and grading

Thurs
- Abstraction, reference, and synthesis
- Procedures and processes
- The procedure as black box
- Procedural abstraction

Week 2

Tues
- Function evaluation
- Substitution model of evaluation
- Abstract data types
- Lists: the fundamental data structure

Thurs
- Conditionals: making procedures flexible
- Recursion: making procedures repetitive
  - linear recursive processes vs. linear iterative processes
  - the "shape" of a process
  - tail recursion and optimizing compilers

Week 3

Tues
- More recursion
  - tree recursion
  - recursion templates with examples
- The myth of efficiency
  - efficient programs v. efficient programmers

Thurs
- Lambda functions: procedures without names
- Procedures as data
- Apply and funcall
- Applicative programming

Week 4

Tues
- Data abstraction: an introduction
- Hierarchical data structures
  - association lists
  - property lists and hash tables
  - trees
  - graphs
  - directed graphs
- Tradeoffs between hierarchical and linear representations

Thurs
- Midterm Exam 1

Week 5

Tues Traversing hierarchical data structures
Dumb search
depth-first with backtracking

Thurs States and problem spaces
Heuristic search
Game search

Week 6

Tues Assignment and local state
Local vs. global state variables
Costs and benefits of variables and assignment

Thurs Iteration: another approach to repetitive procedures

Week 7

Tues Scoping
lexical vs. dynamic scoping
how scoping impacts assignment
call by value vs. call by reference
The lexical closure

Thurs Midterm Exam 2

Week 8

Tues Network representations
relationship to directed graphs
information contained in relationships
Frames and inheritance
extracting inherited information
relationship to objects and OOP
More dumb search
breadth-first search
tradeoffs between breadth-first and depth-first

Thurs Simple object-oriented programming in LISP
The Common LISP Object System (CLOS)

Week 9

Tues Applicative programming revisited

Thurs Macros

Week 10

Tues Building languages on languages
The mini-Scheme interpreter
The LISP evaluator: read, eval, print

Thurs Production systems
rule-based programming
forward- and backward-chaining
Logic, predicate calculus, and deductive inference
Logic programming and Prolog

References

Abelson, H., & Sussman, G.J. (1985). *Structure and Interpretation of Computer Programs.* New York: McGraw-Hill, Inc.

Barr, A., & Feigenbaum, E.A. (1981). *The Handbook of Artificial Intelligence: Volume I.* Los Altos, CA: William Kaufmann, Inc.

Boden, M.A. (1977). *Artificial Intelligence and Natural Man.* New York: Basic Books, Inc.

Charniak, E., & McDermott, D. (1985). *Introduction to Artificial Intelligence.* Reading, MA: Addison-Wesley Publishing Company.

Slagle, J.R. (1971). *Artificial Intelligence: The Heuristic Programming Approach.* New York: McGraw-Hill, Inc.

Winston, P.H. (1992). *Artificial Intelligence* (third edition). Reading, MA: Addison-Wesley Publishing Company.