Introducing Computer Science Using a Breadth-First Approach and Functional Programming

Scott Vandenberg
Computer Science Department
Siena College
Loudonville, NY 12211
vandenberg@siena.edu

Michael Wollowski*
Computer Science Department
Rose-Hulman Institute of Technology
Terre Haute, IN 47803
wollowsk@cs.rose-hulman.edu

Abstract

We present a breadth-first, lecture- and lab-based approach to introducing Computer Science that uses functional programming. Functional programming provides a low-overhead introduction to programming (no types, few constructs, and little syntax), enabling students to write, in their first semester, programs sophisticated enough to exemplify important concepts of Computer Science. It also encourages good programming style (modular design and testing, e.g.) and serves as an introduction to an important problem-solving paradigm. The course gives the students a broad overview of Computer Science and helps them gauge their interest in the field.

1 The Challenge

Three years ago, the Computer Science department decided to add a new introductory course in Computer Science to precede the existing first course, a pure programming course. The new course, which we describe here, is the first course taken by Computer Science majors and minors and satisfies a College core requirement in quantitative reasoning. We had several goals in mind when designing the course: to give all students (majors, minors, and students taking the course to satisfy a core requirement) a broad overview of the field of Computer Science; to teach them good programming habits early in their programming career; to enable students to write good, useful programs in their first semester; and to expose the majors and minors to as many programming paradigms as possible throughout the curriculum.

The course is taken by students with a wide variety of programming backgrounds and experiences. Computer Science majors account for roughly 25% of the enrollment, minors for 25%, and students who take it to satisfy the core or other requirements make up the remainder. An important challenge was to choose a software environment that would enable us to meet all the goals listed in the previous paragraph while serving a student body with a wide variety of backgrounds in programming; similar challenges but different approaches are presented in [3]. The course format is two lecture hours and two lab hours per week. The lectures take place in a classroom with a networked computer connected to an overhead projection system. The computer is configured in the same manner as those in the labs. In the labs, our students work in pairs, with a faculty facilitator to guide them through the experiments.

2 The Approach

The purpose of this course is to introduce our students to the discipline of Computer Science as an activity that uses a general-purpose problem-solving machine. They learn about the physical (hardware) characteristics of the machine and the ways in which information, problems, and solutions (software) are represented in the machine. The course is designed so that it enhances our students' problem-solving abilities by employing techniques such as problem reduction, modular design, and testing. These techniques can be applied in many settings both inside and outside of Computer Science.

We selected the text *An Invitation to Computer Science* by G. Michael Schneider and Judith L. Gersting [4]. This text provides a broad introduction to Computer Science and comes with a lab manual and lab software that illustrate fundamental Computer Science concepts in an interactive, hands-on manner. The text does not rely on a particular programming language or paradigm. This was important as we wanted to use Scheme as a language for the course (see Section 2.2). We chose Chez Scheme [1] and installed

* This work was done while the author was at Siena College.

it on our departmental server. Chez Scheme is an efficient production version, with customer support and good tracing and debugging facilities to help the novice programmer. Students can connect to it using X-Windows or Telnet, from any internet location supporting those modes of connection. We make syllabi, course assignments, solutions, and review exercises available on a course web page.

As we shall illustrate below, a breadth-first approach and the use of a functional programming language dovetail quite nicely into a fairly comprehensive introduction to Computer Science that contains a healthy balance of theory and practice. The course design is based partly on ideas from Computing Curricula 1991 [5] and from a model curriculum for a liberal arts Computer Science degree [6]. While not a programming course, programming is used to educate our students about the field of Computer Science. Our general approach is to cover, each week, certain aspects of Computer Science and functional programming, and wherever possible show their connection.

For example, when investigating language definition and translation and the role BNF grammars play in them, we have our students implement part of a lexical analyzer for Scheme. The correspondence between rules of the BNF grammar and the Scheme functions is striking and is a good demonstration of how theory and practice support each other (see Figure 1).

<ident> ::= <initial_char><subsequent>

<subsequent> ::= Λ | <sub_char><subsequent> | <sub_char>

<sub_char> ::= <initial_char> | <digit> | . | + | -

```
(define (ident char_list)
  (and (initial_char (first char_list))
       (subsequent (rest char_list))))

(define (subsequent char_list)
  (cond ((null? char_list) #t)
        ((and (sub_char (first char_list))
              (subsequent (rest char_list))) #t)
        (else #f)))

(define (sub_char char)
  (or (initial_char char)
      (digit char)
      (eq? char #\.)
      (eq? char #\+)
      (eq? char #\-)))
```

Figure 1: Partial BNF and Scheme for Lexical Analysis

The BNF rules in Figure 1 are adapted from one of our lab exercises and are part of the full BNF for a Scheme identifier. Each Scheme function in the figure implements the corresponding rule in the BNF. (Note that we use "first" and "rest" in place of "car" and "cdr", and that we have omitted several of the BNF rules for brevity.) In this exercise, the students are given the BNF rules and are asked to write the Scheme functions. Each recursive BNF rule becomes a similarly recursive Scheme function and each disjunct in a BNF rule becomes a disjunct in a Scheme function (although we can, in the case of the "subsequent" function, avoid a direct translation of the third disjunct in its BNF rule).

2.1 The Breadth-First Nature of the Course

Before introducing this course, our first Computer Science course was a standard programming course in which we used Pascal. With such a course, students got the impression that Computer Science is just programming. Instead, we wanted to give our students a broad introduction to the major areas of Computer Science, hoping that this may serve as a framework and context on which to build for subsequent courses.

In many curricula, including our old curriculum, there is no single place in which students could construct a picture of Computer Science as a discipline. While a curious student can piece together an idea of Computer Science over a period of several years of studying Computer Science, such a process assumes that a student takes a very wide variety of courses. In practice, this assumption does not always hold. By giving a broad introduction to Computer Science, we ensure that our majors are at least introduced to important Computer Science concepts, such as language translation, hardware design, history of computing, alternative programming paradigms, etc, which are covered only in upper level electives.

Furthermore, our students, like those at many schools, were learning recursion late in their second semester then promptly forgetting this valuable problem-solving approach, in part due to the recognized difficulty of introducing it in a procedural language [2]. Additionally, the functional programming paradigm was only seen by students if they took a programming language or Artificial Intelligence course, whereas the procedural and object-oriented paradigms were, and still are, covered very thoroughly later in the curriculum.

This course is a prerequisite for all our major courses and students typically do not test out of it. Anyone who has AP credit typically receives credit for our second course. In the following subsection we motivate the choice of the functional programming paradigm. Following that, we give a brief outline of the course and describe a specific laboratory as an example of our breadth-first approach to Computer Science with functional programming.

2.2 Choice of Programming Paradigm and Language

The choice of Scheme was motivated by a frustration over our students' programming habits. Submitted programs tended to contain lengthy procedures (over a page long) and hence were difficult to debug and grade. Other efforts [7] have recognized the need for an appropriate instructional environment, and our choice of Scheme in a breadth-first, lab-based course provides such an environment.

In Scheme, the most natural way to program is by designing small functions, something on the order of one recursion per function. This means that top-down design principles have to be strictly followed. While this design principle was taught even when using Pascal, in Scheme the temptation to violate this principle is much lower than in procedural languages. Speculating on the reason for this, it seems that (i) it takes a certain amount of practice to learn how to set up a recursive function, hence preventing overuse and (ii) it is very easy to set up an additional function and add a call to it in the function under development.

Intertwined with problem reduction and the implementation of programs in terms of many small functions is incremental testing. A benefit of Scheme is that the programmer does not have to spend valuable time declaring trivial variables and data-structures. This not only saves time but also enables students to easily create test data for individual functions, without putting them into a larger cluster of functions that share complex data structures. Given this, we set up lab exercises so that our students indeed test and debug their programs incrementally.

The consequent use of top-down design and incremental testing were the primary reasons for choosing Scheme. There were, however, other important reasons. The syntactic simplicity of Scheme enables our students to focus on design issues rather than on the intricacies of the syntax and type declarations of procedural languages. This in turn facilitates the writing of algorithmically interesting programs fairly early in the course.

A pleasant side-effect of the use of Scheme is that almost none of our freshmen has used or even heard of the language. This course satisfies the college core requirement for quantitative reasoning, and as mentioned earlier, the majority of the students in this course are not Computer Science majors or minors. The use of Scheme acts as a great equalizer, where a diligent humanities student can productively participate in class as much as a Computer Science major with a good amount of programming experience.

Since Scheme requires the use of good design techniques, an introductory course can be set up as a course that emphasizes, among other things, problem solving methods, starting with a general statement of a problem, down to a finished program, concentrating on problem reduction methods such as top-down design and recursion. This makes it a useful course even for those students who take it to satisfy the core. By the end of the semester, our students can implement complex and sophisticated programs. Some examples from previous semesters include Eliza and Blackjack.

3 Putting it all Together

Figure 2 contains a course outline, listing each week's Computer Science and Scheme topics. Each week, we tightly integrate the general Computer Science material and the Scheme material, as exemplified in Section 2 with the BNF exercise. The remainder of this section is a more extended example of a typical laboratory session.

Each lab session, all of which were written by the authors, is split into six sections: *goals*, *background readings*, *system know-how*, *expanding general Computer Science*

WEEK	CS TOPICS	SCHEME TOPICS
1	Introduction to CS and algorithms	System Introduction (OS, Windows, UNIX, emacs)
2	Algorithm Design and Examples	Scheme Arithmetic and Simple Function Calls
3	Information Representation	Lists and Basic List Operations
4	Boolean Logic	Boolean Functions
5	Circuits; Modular Design and Testing	Function Composition; Testing; Debugging
6	von Neumann Machines	Recursion; Tracing Function Calls
7	Algorithm Efficiency	Multiple Algorithms for the Same Problem; Accumulators
8	System Software; Virtual Machines	Implementing System Software (e.g. print queues)
9	Abstraction and Programming Paradigms	Generalization; Functions as Parameters (e.g. encryption funcs.)
10	Language Translation	Deep Recursion; Implementing a Scanner
11	Computability; Turing Machines	Functions to Implement Specific Turing Machines
12	Imperative Programming	Let (assignment) and Do (iteration)
13	Catch Up and Review	Completion of an Implementation Project

Figure 2: Course Outline (weeks are approximate and expand slightly to a 14-week course)

knowledge, developing programming solutions in Scheme, and *short essays* challenging students to think critically and creatively about issues related to the lab topics. Labs build on knowledge and experience from prior labs as well as on lecture materials. In addition to Scheme, the labs use the software that comes with the textbook. This software enables experimentation with many fundamental Computer Science concepts and visualization of how some aspects of a computer work. Among others, the textbook's lab manual and software includes simulators for circuit design, a von Neumann machine, an assembler, and a Turing Machine.

To give a sense of the experimental and collaborative nature of the lab sessions, we now describe an actual and representative lab assignment. It is the third lab session of the semester (usually held during the fourth week of classes) and its goals are to design and test simple electronic circuits and to practice Boolean logic and conditionals in Scheme.

The background for this lab comes from the text, from lectures, and from handouts. The system know-how section of this lab covers some features of the editor used to compose programs and text, a brief introduction to an electronic mail program that is commonly used on campus, and some features of the windowing system software that can be used to cut and paste text. The fourth section, on general Computer Science knowledge, uses the circuit simulator software provided by the textbook. Students will spend 30-45 minutes tracing, modifying, and then developing from scratch their own circuits. Figure 3 is a circuit produced by a student using this software. This circuit is a complicated version of the Boolean function NAND which is false if and only if both its inputs are true.

These experiments with Boolean logic in a hardware setting are followed, in the lab's next section, by a study of Boolean logic from the programming perspective. Students are asked to implement Scheme functions that simulate some of the circuits developed in the previous section of the lab. Figure 4 is a Scheme function that produces the same output as the circuit in Figure 3. This Scheme function is derived from the circuit by a straightforward translation of gates into equivalent Scheme functions.

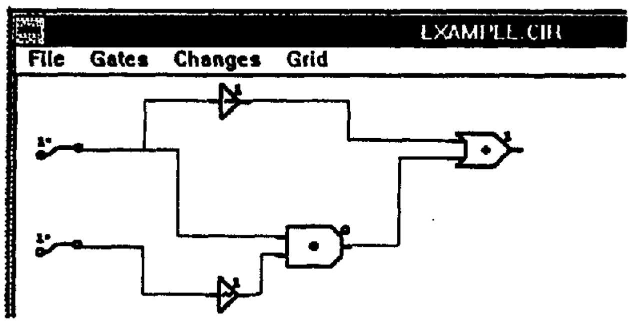

Figure 3: A Circuit Drawn in the Simulator

```
(or (not a)
    (and a
         (not b)))
```

Figure 4

```
(not (and a b))
```

Figure 5

Figure 5 is a more concise implementation of the circuit in Scheme. This implementation requires a closer examination of the truth-table for the circuit. A more efficient version of the circuit itself could have been constructed in the same manner. By using the circuit simulator and Scheme software in this way, students see the relationships between the hardware and software aspects of Boolean logic.

In the last section of the lab, students are asked to write short essays based on their experiences with the lab software. Since they have been working with a system of 2-valued (true/false) logic, we ask them to speculate on the advantages and drawbacks of a 3-valued system of logic, assuming the underlying hardware is based on devices with three, rather than two, easily distinguished states (0, 1, and 2 instead of just the 0 and 1 of binary logic).

The lectures and labs are tightly integrated: many topics are introduced in lecture with online demonstrations and experiments that are continued in lab. For example, to prepare for the lab described above, we review Boolean logic and introduce the concepts of gates, circuit design, and conditional statements in Scheme. During this lecture, we typically use and experiment with truth tables to illustrate the operations of Boolean logic and to assist in circuit design. Truth tables serve to specify circuits. From truth tables, circuits can be constructed in an algorithmic manner. Concurrently, we experiment with conditional statements in Scheme and illustrate the use of Boolean logic in Scheme. Furthermore, we explain the details and subtleties of the circuit simulator, enabling students to concentrate on the important issues of design when they are in lab.

4 Experiences

Students have received a broad introduction to Computer Science, strengthened by their hands-on experience with Scheme and the lab software. The simplicity of Scheme's syntax allows us to spend half the course time on Computer Science concepts and allows us to examine the close relationship between programming and Computer Science.

Students are writing complex, sophisticated programs by the end of their 1st semester, and are doing so in a disciplined way, using the design techniques introduced in the course and encouraged by the software. This contributes to a sense of accomplishment. It would have been more difficult if we had used a procedural language due to the complexity of the syntax and the time spent learning it. The highly available nature of most of the course software has enabled students to work on their homeworks and labs assignments (when not completed during the allotted lab time) from any location. The introduction of recursion has made this topic easier to teach when it is covered in procedural languages later in the curriculum.

The experimental and collaborative nature of the lab exercises stimulates active discussions within and between the student pairs and between students and the faculty instructors. The students feel comfortable approaching other groups and the lab instructor to discuss their ideas. The lab exercises also provide ample opportunities for both oral and written communication about Computer Science. It is our students' communication skills, combined with their technical skills, that make them effective Computer Scientists after graduation. From the faculty's point of view, the labs give instant feedback about our students' thinking so that lectures and assignments can be tailored to the current strengths and weaknesses of the students.

While we chose Scheme primarily for didactic reasons, we have come to appreciate the fact that almost nobody has used Scheme in high school. This makes the language, and indeed the entire course, a new experience for all students: non-majors do not feel "left out" by not knowing the programming paradigm or language, and experienced students learn a new language and a broad overview of the field of Computer Science, which they do not get in most high-school (or college) programming courses.

5 Conclusions and the Future

The course has proved to be challenging. Nevertheless, enrollments have increased steadily and the course has succeeded in attracting new majors to Computer Science. The course gives students an opportunity to assess their interests and abilities in the field of Computer Science as a whole, not just in programming. After four semesters of teaching the course we are confident, based on our experiences and on student evaluations, that it is an excellent introduction to Computer Science. We have demonstrated that functional programming can be used in a lab-based, breadth-first environment to substantially enrich the depth and breadth of topics in a first course in Computer Science. This particular mix of topics has worked well for us, but other general Computer Science topics could be substituted for some of these without greatly affecting the value of the course, as long as proper support for those topics is in the textbook and/or a simulation.

The course satisfies the College's quantitative reasoning core requirement by involving quantitative analysis of physical and logical systems and a healthy dose of symbol manipulation, an important aspect of both mathematics and Computer Science. Symbols are stored in the computer as quantities, and in Computer Science we certainly reason about these quantities. In addition, all core courses at the College must have a writing component; in this course, the students write short essays as part of the labs and, of course, write programs.

In the future, we plan to make the lab materials available on the world-wide web. We will also be using a new version of Chez Scheme [1] that has support for graphics. This will make some lab exercises more appealing to some students.

References

[1] Chez Scheme. http://www.scheme.com

[2] Ginat, D. and Shifroni, E. Teaching Recursion in a Procedural Environment - How much should we emphasize the Computing Model? In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (March 1999), ACM Press, 127-131.

[3] Lewandowski, G. and Morehead, A. Computer Science Through the Eyes of Dead Monkeys: Learning Styles and Interaction in Computer Science I. In *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education* (February 1998), ACM Press, 312-316.

[4] Schneider, G.M. and Gersting, J.L. *An Invitation to Computer Science* (2nd ed.). Brooks/Cole, Pacific Grove, CA, 1998.

[5] Tucker, A.B., Barnes, B.H., Aiken, R.M., Barker, K., Bruce, K.B., Cain, J.T., Conry, S.E., Engel, G.L., Epstein, R.G., Lidtke, D.K., Mulder, M.C., Rogers, J.B., Spafford, E.H., and Turner, A.J. Computing Curricula 1991. *Commun. ACM 34*, 6 (June 1991), 68-84.

[6] Walker, H.M. and Schneider, G.M. A Revised Model Curriculum for a Liberal Arts Degree in Computer Science. *Commun. ACM 39*, 12 (December 1996), 85-95.

[7] Ziegler, U. and Crews, T. An Integrated Program Development Tool for Teaching and Learning How to Program. In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (March 1999), ACM Press, 276-280.