Reconsidering the Elementary Courses in Computer Science

Richard A. Brown
Department of Mathematics
St. Olaf College
rab@stolaf.edu

Abstract

A collection of four *foundation courses* is proposed for introductory computer science (CS) at a liberal arts college. A breadth-first, multi-paradigm CS1 course serves as the prerequisite for each of the remaining three foundation courses. This CS1 course, which has no prerequisites of its own, focuses on recurring concepts in CS and development of abstraction and analytical thinking skills, made tangible through Scheme programming exercises, and simultaneously serves CS majors and non-majors. A C++-based course in software design and implementation develops programming skills, introduces notions of software engineering, and includes a waterfall-model team software project. A standard computer organization course and a Scheme-based mathematics course complete the collection. These courses address various national curricular recommendations and satisfy local goals for CS in a liberal arts college.

Introduction

A liberal arts college faces many challenges when designing and implementing elementary courses in Computer Science (CS). National curricular recommendations from the CS professional societies such as *Computing Curricula 2001* ("CC2001", 2001) seem oriented toward universities, which typically house much larger scale operations. Liberal arts institutions place their values in different directions, e.g., expecting small major programs that leave over half of a student's total credit load for coursework in non-major courses. Small, often private, undergraduate-only schools face very different practical constraints than large universities with graduate students (Walker and Schneider, 1996). Even though liberal arts institutions can certainly meet these challenges through standard introductory sequences, there remain many choices to make. For example, CC2001 identifies six distinct *approaches* to the introductory course CS1, each of which has many realizations in terms of text and syllabus ("CC2001", 2001, Appendix B). Most of these approaches adopt a "depth-first" strategy, which might not seem optimal for an institution that wants to offer a counterpart of the survey courses commonly offered as introductions to other disciplines. How can a college best match its early CS courses to its local objectives and campus context?

Many colleges have developed new elementary courses in their searches for responses to these challenges. For example, some institutions have introduced locally developed "prequels" to CS1, often called "CS0" courses, that present a broad image of the discipline of CS and/or appeal to students who might not otherwise take a CS1 course. These courses have enjoyed considerable success at some liberal arts colleges. However, the effectiveness of a CS0 course depends heavily on local factors. Does a particular version of a CS0 course draw students who are unlikely to continue in CS? If the CS major does not require CS0, what incentive will CS0 students have to continue in CS? If CS0 becomes required for majors, perhaps providing them with a valuable initial overview of the discipline, can the number of required courses be increased while respecting the liberal arts context of the CS major, or must another requirement be dropped? This example of CS0 illustrates the complexity and difficulty of finding elementary CS courses that meet the manifold goals of a particular liberal arts college.

St. Olaf College has evolved its elementary course offerings over a fifteen-year period, arriving at a non-traditional collection of courses that addresses local institutional goals, including compactness and flexibility of the major program, access for non-majors, a breadth-first introduction to the discipline, and a good institutional fit with our liberal arts environment. We hardly claim that our approach would work at all colleges, but we hope that parts of our model may help other institutions as they seek better and better elementary courses for their own unique college contexts.

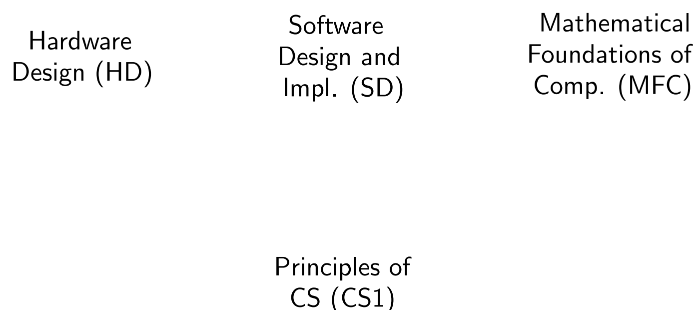

Figure 1: Foundation courses in CS at St. Olaf.

Institutional context

St. Olaf is a private undergraduate college enrolling approximately 2800 students, located in Northfield, Minnesota. A college of the Lutheran Church, St. Olaf is known for strength in academics, notably in Music, Mathematics, and the natural sciences, and for international and off-campus study. The Departments of Mathematics and Physics initiated an interdisciplinary concentration in CS in 1974 under the direction of Richard Allen. From the outset, the CS concentration emphasized both conceptual principles of CS and hands-on application of those concepts. The program's carefully chosen example systems have heavily influenced both the program and its students. For example, St. Olaf CS was among the earliest adopters of the UNIX operating system west of the Mississippi (1974), a fact that urged our students toward early strength in C programming and later involvement in the open source software movement, including Linux. By the early 1990s, the CS concentration was no longer an interdisciplinary program, now requiring six courses specifically in CS. Local fiscal concerns prevented expansion to a CS major until a pivotal program review in 2002; our first three CS majors will graduate in 2004.

The structure of the *foundation courses* in St. Olaf's CS major (see Figure 1) was developed during the days of the smaller concentration, and that structure for elementary courses has served us well as a basis for a full CS major program. Instead of an introductory sequence, we offer a four-course "introductory tree" consisting of a breadth-first CS1 course followed by a choice of second courses which may be taken in any order. Each second course expands on an aspect of the discipline: *Hardware Design (HD)*; *Software Design and Implementation (SD)*; and *Mathematical Foundations of Computing (MFC)*. CS1 has no prerequisites, although we advise less prepared first-year students to get a term of college experience before taking CS1. The CS1 course is the sole prerequisite for each second course, although MFC effectively calls for "mathematical maturity" at the general level of a calculus course, a reasonable assumption at St. Olaf.

```
(define make-account
  (lambda (init-bal) ;; state variable:  initial balance for this account
    (let ((bal  init-bal))  ;; state variable:  current balance for this account
      (lambda (method args) ;; method name (symbol), list of args
        ;; NOTE: a function returned by this lambda is an account object
        (case method
          ((show-balance)  bal)
          ((show-initial-balance)  init-bal)
          ((deposit)  (set! bal (+ bal (car args)))
                      bal)
          ((withdraw)  (set! bal (- bal (car args)))
                       bal))))))
```

Example calls:

```
(define acct1 (make-account 500.00))
(send acct1 'show-balance)    ;; Return value:  500.
(send acct1 'deposit 150.00)  ;; Return value:  650.
(send acct1 'show-balance)    ;; Return value:  650.
(send acct1 'show-initial-balance)  ;; Return value:  500.
(acct1 'show-initial-balance ())    ;; Return value:  500.
```

Figure 2: Object-oriented programming. First, we define a constructor for objects in the class Account, representing a simple bank account. Then, in the example calls, we construct an Account object `acct1`, and call several methods using a function *send* (definition omitted here). The final line shows how to call a method without `send`.

CS1, Principles of Computer Science

St. Olaf did not create a "CS1" course until 1989, near the end of Pascal's dominance as an introductory programming language. Our CS1 has used the Scheme language from the beginning, at first following (Ableson et al., 1985), which provides an outstanding education in CS. However, since that book's underlying mathematical expectations seemed too ambitious for many of our liberal-arts audience students, we switched to (Springer and Friedman, 1989) early on. We soon found ourselves supplementing this text with presentations and exercises on documentation, invariants, etc., then with a different approach to teaching procedural abstraction, an introduction to object-oriented programming concepts, and more, until the course had evolved during the period 1992-1998 into a new entity with its own problem sets and manuscript text.

insert-left

3 Arguments: Two Scheme values and any Scheme list.
Return: A list consisting of all elements of *arg3* in order, except with each occurrence of *arg1* immediately preceded by a new occurrence of *arg2*.

Example call:
`(insert-left 'a 'z '(a b c a)) ;Return value:  (z a b c z a)`

```
(define insert-left
  (lambda (pattern insert lis) ; two Scheme values, any list
    (letrec ((helper
              (lambda (ls done)
                ;; ITERATION INVARIANT:
                ;; list ls holds all values of lis not yet seen
                ;; list done holds elements of lis already seen,
                ;;  after inserting  insert  just before every
                ;;  occurrence of  pattern, in the original order
                (cond
                 ((null? ls) done)
                 ;; assert:  there is at least one unseen element
                 ((equal? (car ls) pattern)
                  (helper (cdr ls)
                    (append done (cons insert (cons pattern ())))))
                 ;; assert:  first element in ls differs from pattern
                 (else (helper (cdr ls)
                          (append done (cons (car ls) ()))))))))
      (helper lis ()))))
```

Figure 3: An example of documentation, tail recursion, and reasoning about program correctness. The (iteration) invariant and the "asserts" outline a proof of the correctness of an `insert-left` function definition satisfying the boxed specification.

In its current form, we describe our CS1 course as an introduction to the way computer scientists think. We present new concepts in a strategic order; each daily step is simple but subtle, with daily homework designed to check that new subtlety. Throughout the term, students develop better and better facility with abstraction, as well as improved analytical thinking skills. Although it is difficult to quantify these qualities of mind, we find that investing heavily in these thinking skills in CS1 provides our students with "intellectual headroom" that serves them well in later CS courses. For example, nonlinear recursion and procedural abstraction become "old hat" before their second CS courses, and a quick Scheme-based example suffices to jog their memories for applications in new contexts.

Our CS1 is a "three paradigm course" that spends roughly equal time on functional programming, imperative programming, and object-oriented programming, in that

order. Thus, we present recursion before iteration, a natural ordering for newcomers to programming and an interesting twist for experienced programmers. Scheme has no standard loop constructs, but one can program iteratively using tail recursion (see Figure 3); this represents both iterative algorithmic thinking and actual iterative computation, since Scheme is a properly tail-recursive language. Scheme closures enable students to experiment with object-oriented programming, learn the associated terminology, and explore notions such as inheritance using an object mechanism they literally build themselves.

Despite the use of Scheme language, we view this CS1 as a *breadth-first* introductory course, not a "functions first" course, in the terminology of ("CC2001", 2001, Appendix B). Although we do not systematically survey subject areas in CS, we include fundamental ideas from many areas throughout the course. For example, we introduce the system stack to describe the computational difference between iteration and (non-tail) recursion; we speak of machine addresses and pointers when examining the memory effects of making assignments to alter Scheme's built-in linked lists and underlying `cons` pairs (nodes); we count the number of operations performed when comparing iteration and recursion, and compare the memory storage strategies of lists and Scheme vectors (like arrays, but with an arbitrary type for each element); we emphasize documentation and introduce the notion of provably correct programming by providing (English-language) specifications for functions and demanding invariant assertions ("asserts") from early in the course (see Figure 3); later, we add (loop) invariants for iteration via tail recursion (Figure 3); etc.

Emphasis on principles provides another reason for viewing this CS1 as a breadth-first course. We adopt the "recurring concepts" of ("CC1991", 1991, Section 5.4) as our collection of principles. The CC1991 task force identified fourteen recurring concepts, such as "conceptual and formal models," "levels of abstraction," and "tradeoffs and consequences," which occur throughout the discipline, have a variety of instantiations in multiple subject areas, and have a high degree of independence of particular technologies. To quote the task force,

> Recurring concepts are significant ideas, concerns, principles and processes that help to unify an academic discipline at a deep level. An appreciation for the pervasiveness of these concepts and an ability to apply them in appropriate contexts is one indicator of a graduate's maturity as a computer scientist or engineer... Additionally, these concepts can be used as underlying themes that help tie together curricular materials into cohesive courses. (Section 5.4)

We comment on these principles at the ends of text chapters and in the classroom, and they inform our course's choices of content. We have taken them as a guide, insuring that our course presents a deep view of the discipline, and developing further

“intellectual headroom” in our students at the outset of CS study.

Although this CS1 may sound intimidating, we find that students of any major can succeed in this course. Scheme’s simple syntax, supported by a language-sensitive editor (`emacs`), helps non-CS students get started with programming technology and focus on the concepts more than the language expressions. Staffing a computing laboratory at night with experienced CS students, together with instructor office hours, has provided adequate support (we teach the course without a closed lab). But the most essential key to success is daily homework applying and reinforcing each new concept. We ask students to work out each solution logically, *by hand*, writing out every symbol of each language expression and seeking thorough understanding, *before* entering that solution at the computer and testing it. We make this by-hand work an expected part of the homework submission. Of course, some students do not follow this regime carefully, but those who do gain very thorough understanding, and invaluable sense of confidence and mastery over this material, whatever their background and interests. On the other hand, those who fall behind on homework almost always get lost in the accumulation of small but subtle daily steps, suddenly finding themselves feeling as if they have no idea what is going on in the course—even whose with prior experience in CS. Such “lost” students can salvage their understanding and confidence by going back to the earliest topics where they began to neglect a thorough understanding, then to master that (and subsequent) material, for example, through more disciplined by-hand-first work. The author knows of no other course in any discipline that depends so heavily on daily homework: so seemingly impossible for almost anyone who doesn’t do the homework; and so tractable and confidence-building for almost anyone who keeps up with the daily work, including those with no prior exposure to programming or CS.

SD, Software Design and Implementation

Although we introduced a CS2-like course in early 1993, the second course’s focus meandered until we developed the current *Software Design and Implementation* during the five-year period 1998-2002. As the name suggests, SD emphasizes elements of software engineering; a subsequent course *Algorithms and Data Structures* includes many topics that might commonly be found in a CS2 course.

For many reasons, we have students program in C++ language for the SD course. We eagerly want our students to see CS1 principles and thinking patterns in a contrasting language, so they may know they weren’t just about Scheme. C++ forces students to grapple with syntactic complexity, a necessary skill; emphasizing concepts known since CS1 helps keep syntax from becoming the overwhelming focus. We choose C++ rather than C for the object support; we choose C++ rather than Java so we can explore lower-level issues more concretely. In particular, we emphasize memory management, including local, dynamic, and static allocation and proper deallocation,

and we incorporate significant programming with pointer variables, building on earlier programming with `cons` pairs in CS1. Once students have programmed in both Scheme and C++, they can readily "come up to speed" in many other programming languages, having seen two relative "extremes" among high-level languages.

SD is our only CS course that includes a weekly "closed" laboratory meeting. The syllabus of SD has three parts, roughly equal in length: a transition from Scheme to imperative programming in C++; a succession of C++ classes, exploring memory storage alternatives and relationships between classes; and a team project using a waterfall-model software life-cycle. The first part naturally focuses more on basic issues, including loop statements, programming with pointers, references, and `const`, parameter passing options, etc. The second part begins the course's software engineering emphasis: each day's assignment calls for implementation of a new class using that day's new technique or memory strategy, according to a standardized structured specification of that class. The team project involves many forms of writing: a user manual (which serves as an informal requirements specification); UML-based design decisions for class relationships and roles and for key algorithms; specifications for each class adhering to the course's standard; etc. Each team is responsible for assembling these documents on a web site; students produce many of the documents (e.g., class specifications) using locally developed XML document types.

We have recently added a writing-based four-stage ethical analysis of the project following the ImpactCS grid analysis model (Huff and Martin, 1995). We provide XML-based templates for these documents, and plan to move this ethics work to the second part of the course, to compete less with the project and to give students experience with the mechanisms of our local XML document system in advance of the project itself. This adds a context of ethics, social issues, and professional responsibility to the software design work, and prepares the way for serious study of computing ethics later in the CS major.

From a practical viewpoint, the biggest challenge in designing SD was creating a transition from Scheme to C++ that took direct advantage of the CS1 experience. For example, the first lab and subsequent homework assignments now build on a simple (provided) implementation of Scheme-like lists (of strings), developing a library of C++ functions patterned after Scheme counterparts, with "iteration" accomplished at first through tail recursion. Also, we consistently introduce new topics in the context of prior Scheme work. Given this Scheme-transition need and the low-level emphasis on software engineering, we are unable to find a textbook that truly fits our course. Thus, we have developed a substantial body of online notes, examples, and exercises; we often supplement this with a C++ language reference such as (Deitel and Deitel, 2003).

The primary challenge of SD is the extensive time required for the daily work. We view SD as the most time-intensive CS course we offer, and advise students to avoid

taking other time-consuming courses during the same term as SD. We would rather not have a course with such substantial time-management issues so early in the CS curriculum, but the overall strategy of developing solid programming and project skills in a second course has worked well for later CS courses. Plus, CS1 and SD together provide a valuable two-course package for non-majors and young majors seeking internships.

HD, Hardware Design

Our *Hardware Design* course is a relatively standard computer organization course based on (Tanenbaum, 1999). Computer Organization was the original CS course at St. Olaf, as at many other schools. We have used Tanenbaum's layered approach to computer design for about 15 years, often updated with supplemental material as needed (e.g., with information about newer processors as an edition ages). The multilevel layers of architecture present a compelling abstraction that appears in many other contexts in computing, such as windowing software and network protocols (recall that "levels of abstraction" is a CS1 "principle"). We spend a short time on network protocols at the end of the course to show an alternative layered architecture, as well as to introduce some valuable concepts and terminology of computer networking.

As our foremost goals for HD, we want every student to obtain a conceptual understanding of how a computer works, and to receive a broad exposure to the vast body of computing terminology related to hardware. For instance, we examine a representative example microprogrammed machine in considerable detail, then consider successive enhancements leading toward the microarchitecture of a Pentium chip, in order that students may understand microarchitecture issues, features, and terminology (Tanenbaum, 1999, Chapter 4). We supplement this segment of the course with discussion of alternative processors such as UltraSparc, and of more recent processors such as Itanium. Note that Tanenbaum offers ample emphasis on RISC principles; thoroughly examining a microprogrammed machine provides both historical and technological perspective, and offers a strong context for considering design tradeoffs in processors such as Pentium, comparisons between architecture choices, etc. Tanenbaum's presentation of the design of multiple actual machines at each architecture layer supports this approach, and also serves our local goal of presenting both conceptual principles and realistic examples.

We include assembly programming in the course in a modest project, e.g., checking input strings for palindromes, using a subset of Java Virtual Machine language and a simulator. We don't expect our students to become professional assembly programmers, but experience programming at this level offers invaluable tangible insight about how computers work.

HD has only CS1 as a prerequisite. These students can readily understand the architectural concepts. The students' greatest struggle concerns reading the book. Part of this struggle is desirable, as throughout the course, students develop a capacity to absorb and manage the necessary mounds of technical terms and details. But Tanenbaum's prose compounds the difficulty of this process. In particular, our students find many of Tanenbaum's exercises confusing and unclear. Over time, we have developed restatements and explanatory annotations for such exercises, and have added exercises of our own.

Although our program has a goal of hands-on experience with principles, we do not currently have the resources for a hardware laboratory associated with the course. Instead, we content ourselves with short programming assignments, e.g., given simulated gates, to program a simulation of a one-bit ALU. Students may implement these small individual projects using a programming language of their choice; the projects tend to demand more from C++ or Java programmers than from Scheme programmers, which appropriately asks more of those with experience beyond CS1.

MFC, Mathematical Foundations of Computing

Our new CS major requires a mathematics course designed for computer science students, satisfying many of the "DS" (Discrete Structures) knowledge units in ("CC2001", 2001, Chapter 5). MFC will serve as the prerequisite for core courses in Theory of Computation and in Algorithms and Data Structures, and it may interest mathematics students seeking a "warm-up" course before Abstract Algebra and Elementary Real Analysis. We are developing this course now, with a first offering in Spring 2005, perhaps using (Gries and Schneider, 1993) for partial text support.

Our strategy calls for building directly on the English-language reasoning in CS1, moving to invariant assertions and iteration invariants stated in formal logical notation, basic logic, and formal proofs of program correctness. Thereafter we plan to proceed to a study of standard proof techniques, such as induction and contradiction, with applications to trees and graphs, sets and relations, basics of counting, and discrete probability. Our version of CS1 (which possesses enough mathematical content to satisfy the College's Mathematics requirement) provides a head start on some of the earlier topics, and the assumed calculus-level mathematical maturity will also contribute to students' success in this course.

Analysis

St. Olaf's foundation courses satisfy numerous national curricular recommendations for CS. For example, Figure 4 shows relationships between these courses and the knowledge units of ("CC2001", 2001, Chapter 5). Each course treats most of the

course	CC2001 knowledge units
CS1, *Principles of Computer Science*	PF1 PF2 PF3 PF4 PL7 SE10
SD, *Software Design And Implementation*	SE1 SE3 SE4 SE5 SE7 SE8 SE10 OS5 PL6 SP3
HD, *Hardware Design*	AR1 AR2 AR3 AR4 AR5 AR6 AR7 AR8 AR9 PL2
MFC, *Mathematical Foundations of Computing*	DS1 DS2 DS3 DS4 DS5 DS6 AL5

Figure 4: Analysis of St. Olaf foundation courses in terms of CC 2001 knowledge units.

required units for a particular knowledge area: Programming Fundamentals for CS1; Software Engineering for SD; Architecture and Organization for HD; and Discrete Structures for MFC. Each course also addresses additional knowledge areas, for example, Programming Language units in CS1, SD, and HD, and Social and Professional Issues in SD. The union of knowledge units included in these foundation courses complements core offerings in the CS major, so that a student who takes all foundation courses and all core courses will encounter nearly every required knowledge unit in CC2001 (Brown, 2004c), (Brown, 2004b).

The foundation courses compare favorably with the 2004 Model Curriculum for a Liberal Arts Degree in Computer Science (Consortium, 2004). The new Model Curriculum calls for multiple programming paradigms in the elementary courses, satisfied in our CS1; it recommends an early course oriented toward software engineering, satisfied by SD; the computer organization core course corresponds to HD; and we are interested to learn that the model curriculum recommends a functional introduction to the first mathematics course, called a "Mathematical Foundations" course (compare MFC)! Our four foundation courses represent an alternative approach to the Model Curriculum's "basic courses," and our major retains the Model's structure of basic courses, core courses, electives and a capstone experience.

An analysis of a syllabus for St. Olaf's CS1 (Brown, 2004e) shows that the course does indeed span the list of "recurring concepts" found in ("CC1991", 1991), supporting our contention that CS1 is a "breadth-first" introduction to CS in the sense of CC2001

(Brown, 2004a).

The foundation courses address numerous practical and local considerations for our liberal arts CS major. Among the practical issues, the compact, tree-like prerequisite structure enables students to access most core courses after as few as two semesters of study (possibly requiring two courses in the second semester) (Brown, 2004d). This benefits students who want to explore CS among other possible major interests, and students who want to schedule a term abroad (common at our College). The four foundation courses constitute a complete set of prerequisites for all core courses. In our CS1, we offer a breadth-first introduction suitable for all majors (see below) and required of CS majors without increasing the total number of required courses for a CS major, as may result from adding a CS0 course. Furthermore, we have found that the "software engineering second" strategy of SD pays off handsomely in subsequent courses, because students emerge from SD possessing experience with a structured team software project, a UML-based object-oriented design, awareness of ethical issues and a grid-based ethical analysis of software, and with considerable programming proficiency in a practical and complex programming language (C++). As the SD course developed at St. Olaf, teachers of core courses soon noticed improved programming and project skills among the students.

Among the local objectives for CS supported by the foundation courses, the breadth-first CS1 serves many goals. Non-majors who take this single course emerge with a relatively comprehensive view of the nature of CS, focused broadly on a survey of concepts, and presenting an image of a computer scientist's disciplinary viewpoint. SD, HD, and MFC all make sensible choices of a second course depending on student interests, and each of these choices constitutes a coherent two-course terminal sequence, exploring software, hardware, or mathematical computer science. The "software" combination CS1—SD provides students with a strong two-course background for internships or interdisciplinary work, as well as for subsequent courses in CS. The experience of learning two languages in this two-course sequence drives home the independence of technology possessed by CS principles, gives students some background in picking up new languages, and encourages open-mindedness toward choices of programming language. SD emphasizes written and oral communication, consistent with the liberal arts; the course can satisfy the College's writing requirement. In addition, the courses CS1, SD, and MFC (with its functional programming combined with mathematics) all promote St. Olaf's philosophy of emphasizing both CS principles and "hands-on" experiences.

The breadth-first, no-prerequisites CS1 invites non-majors to try a CS course, and provides them with access to the major. Figure 5 is drawn from the program review of our former Concentration, a non-major program that preceded our current CS major. This figure suggests that more students from different majors completed a CS concentration as we developed our current approach to CS1 (this development took place during the period 1992—1998). While there may be many explanations for

category	grad years '85-'94	mean	grad years '95-'01	mean	pct.change in mean
Number of grads	91	9.1	78	11.1	+22.0%
Majors earned	105	10.5	93	13.3	+26.8%
Grads with Math majors	73	7.3	38	5.4	-26.1%
Grads without Math majors	18	1.8	40	5.7	+217.7%
Pct of grads w/o Math maj	19.8%		51.1%		
Count of distinct majors represented	13		20		

Figure 5: Majors completed by St. Olaf CS concentrators. Note that a decline in total Mathematics majors generally corresponds to the decline of CS concentrators who completed a Mathematics major.

these changes, the notable increases in the last three rows in Figure 5 indicate that our approach to CS1 did not drive away students in other disciplines. We believe that this CS1 course becomes a way for students who may think of themselves as "non-math people" to make a fresh start in learning analytical thinking.

We do not claim that these foundation courses satisfy all needs for all people. We have recently introduced an accelerated course encompassing the material of CS1 and SD combined, to serve the needs students with prior programming experience comparable to a high school AP course in CS. We have already noted the challenges of heavy time expectations in SD, and may propose additional credit for the required laboratory for that course. The MFC course currently remains in the concept stage, and we anticipate that we will have much to learn from first offerings of that course. Although we have long-term experience with the other three foundation courses, our CS major is new, and its ongoing success remains to be established. Finally, only St. Olaf has offered this package of elementary courses, to our knowledge. Can they thrive on other campuses? Certainly, few colleges are willing to risk radically new approaches to CS1, the primary entry point to the CS major.

Yet, perhaps some elements of these courses will assist other institutions in their search for improvements in the elementary CS courses.

Acknowledgments

Richard Allen, the founder of St. Olaf's CS program and my mentor, has profoundly influenced these ideas, as have my CS teaching colleagues Matthew Richey, Steve McKelvey, and Amelia Taylor. I am especially grateful to Steve as co-author of the CS1 manuscript. Chuck Huff, our resident computer ethicist, is teaching me about ethics and about the grid-based strategy for ethical analysis used in SD. Chuck also gave feedback on an early abstract for this paper. Of course, input from many students has deeply affected the development of these foundation courses.

References

Ableson, H., Sussman, G. J., and Sussman, J. (1985). *Structure and Interpretation of Computer Programs.* MIT Press.

Brown, R. A. (2004a). A breadth-first introduction to CS. Retrieved March 11, 2004, from http://www.stolaf.edu/depts/cs/academics/more/breadthfirst.html.

Brown, R. A. (2004b). CS body of knowledge. Retrieved March 11, 2004, from http://www.stolaf.edu/depts/cs/academics/more/csbok.html.

Brown, R. A. (2004c). Listing of CS courses. Retrieved March 11, 2004, from `http://www.stolaf.edu/depts/cs/academics/courses/list.html`.

Brown, R. A. (2004d). Overview of CS courses. Retrieved March 11, 2004, from `http://www.stolaf.edu/depts/cs/academics/courses/overview.html`.

Brown, R. A. (2004e). Recurring concepts in St. Olaf's CS1. Retrieved March 11, 2004, from `http://www.stolaf.edu/depts/cs/academics/more/cs1principles.html`.

"CC1991" (1991). Computing curricula 1991. Retrieved March 11, 2004, from `http://www.computer.org/education/cc1991/`.

"CC2001" (2001). Computing curricula 2001. Retrieved March 11, 2004, from `http://www.computer.org/education/cc2001/report/index.html`.

Consortium (2004). A 2004 model curriculum for a liberal arts degree in computer science. Draft dated February 27, 2004, distributed at SIGCSE 2004 Technical Symposium, Norfolk, VA., on March 5, 2004.

Deitel, H. M. and Deitel, P. J. (2003). *C++ How to Program*. Prentice-Hall, fourth edition.

Gries, D. and Schneider, F. B. (1993). *A logical approach to discrete math*. Texts and Monographs in Computer Science. Springer.

Huff, C. W. and Martin, D. (1995). Computing consequences : A framework for teaching ethical computing. *Communications of the ACM*, 38(12):75–84.

Springer, G. and Friedman, D. P. (1989). *Scheme and the Art of Programming*. MIT Press.

Tanenbaum, A. S. (1999). *Structured Computer Organization*. Prentice-Hall, fourth edition.

Walker, H. and Schneider, G. (1996). A revised model curriculum for a liberal arts degree in computer science. *Communications of the ACM*, 39(12):85–95.