Science of Computing Suite (SOCS): Resources for a Breadth-First Introduction

Carl Burch
Saint John's University
Collegeville, Minn., USA
cburch@csbsju.edu

Lynn Ziegler
Saint John's University
Collegeville, Minn., USA
lziegler@csbsju.edu

ABSTRACT

Over the last ten years, our department's breadth-first introductory course has evolved independently of other survey courses in computer science. Due to its success, we duplicated the ideas into our course for non-majors, and this has also proven successful. None of the published resources match our vision for these courses, and so the department has developed its own. In this paper, we describe the design of the majors course, and we introduce a variety of resources developed for both courses. These resources, which could be useful in many other courses also, are freely available through the Web.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computers & Information Science Education—*Computer Science Education*

General Terms

Design, Documentation, Theory

Keywords

CS1, breadth-first, simulation

1. INTRODUCTION

In Fall 1993, our department introduced to its computer science major an introductory, breadth-first course, *Introduction to the Science of Computing* (CSCI 150). Despite persistently searching published textbooks over the years, we have found none that closely match our vision for the course. As a result, the department has developed its own resources. Today, this course represents the cumulative work of the nine faculty who have taught the 42 sections of the course offered since then. The authors of the current paper represent the two faculty who have taught the course most recently and who have developed most of the current resources.

SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.

In Fall 1997, the department duplicated the ideas behind CSCI 150 into our course for non-majors, resulting in *Computing: Science and Applications* (CSCI 130). This course has proven very popular with students, and 28 sections have been taught since its inception.

In this paper, we describe the design of CSCI 150, and we introduce a variety of resources developed for both CSCI 130 and CSCI 150. These resources, which include written resources and a suite of support software designed for laboratories, are freely available through the Web.

2. CSCI 150 DESIGN

The CSCI 150 design derives from the particular circumstances of our colleges, as it must.[1] Since the colleges it serves are liberal arts colleges, the department feels that a breadth-first introduction is important. This is an implicit recognition that students in their first year are exploring possible majors; many complete CSCI 150 as a part of this exploration and choose to concentrate on another discipline. For these students, the primary role of CSCI 150 is as a piece of their overall liberal education. Such students should take from the course an understanding of the important ideas in computer science.

Indeed, the colleges have recognized this role by counting CSCI 150 (and, incidentally, CSCI 130) toward their common core curriculum. The courses fulfill the students' "quantitative reasoning" requirement and also act as "natural science" courses. Though these designations are somewhat arbitrary, the ways of thinking they indicate affect the choice of course content, too.

As for many institutions, the 1989 Denning report "Computing as a Discipline" inspires our introductory survey [8]. Our experience with CSCI 150 matches an important observation in the report.

> The introductory sequence should bring out the underlying unity of the field and should flow from topic to topic in a pedagogically natural way. It would therefore be inadequate to organize the course as a sequence of nine sections, one for each of the subareas; such a mapping would appear to be a hodge-podge, with difficult transitions between sections.

One of the biggest issues for a breadth-first approach is exactly this point of coherence. Without some uniting theme,

[1]The department is joint between the College of Saint Benedict and Saint John's University, two independent colleges. Thus, we refer to these host institutions in the plural.

Knowledge unit		CSCI 150 coverage	CC2001 recomm.
DS2	Basic logic	1.4	10
PF1	Fund. programming	7.0	9
PF3	Fund. data structures	2.8	11
AL1	Algorithmic analysis	1.4	4
AL3	Fundamental algorithms	1.4	12
AL5	Basic computability	4.2	6
AL7	Automata theory	4.2	—
AR1	Digital logic	5.6	6
AR2	Representations of data	5.6	3
AR3	Assembly level	4.2	9
OS1	Operating sys. overview	1.4	2
OS3	Concurrency	1.4	6
OS5	Memory management	1.4	5
IS1	Issues in intelligent sys.	1.4	1
IS2	Search	1.4	3

Figure 1: Estimate of CC 2001 knowledge unit coverage.

the course can easily appear to students as a study of independent subjects. This frustrates students, who find that as soon as they have mastered one subject, the course has moved to a very different topic. A unifying theme allows students to build connections between subjects, thus reinforcing their overall understanding.

A survey of past SIGCSE papers indicates that most writers do not describe breadth-first courses in terms of such a uniting strand. Of those that do, the themes identified include the following.

Programming: Students develop small programs related to different concepts in the course during laboratories [15, 14]. For example, when the students study logic, they write programs to compute logical functions (or perhaps even to operate on logic circuits).

Application-oriented: The course explores a variety of concepts surrounding a particular computer application. The two applications mentioned in the literature are the Internet and the Web [10, 12].

Bottom-up: The course moves in progression up the computer systems hierarchy [9, 7, 11]. In other words, the course begins with logic circuits, then digital components, then assembly language programming, then programming in a high-level language.

Algorithms: The course emphasizes algorithms and their efficiency [2]. For example, the course could motivate a unit on logic by the need to express preconditions and postconditions.

While these approaches can work well, our course has developed a very different emphasis: that of *computational models* and their relationship. The course, as currently conceived, includes eight primary units, beginning with a bottom-up approach through the first half. The following describes the topics within the units, along with their relationship to the theme of computational models. (Figure 1 correlates the coverage of topics with the knowledge units of Curriculum 2001 [6].)

Data representation: bits, integer and floating-point representation, multimedia. This unit is necessary to the computational models theme because of data's centrality to the concept of computation.

Circuits: logic gates, Boolean algebra, adders, flip-flops. The logic circuit is one of the course's major computational models.

Machine language: CPU architecture basics, instruction representation, assembly language. Machine language is another major computational model.

High-level programming: imperative constructs, using objects, procedures (all in Java). High-level language is another major computational model.

Operating systems: purpose, multiprocessing, paging. The coverage of multiprocessing is a practical outline of how additional CPUs do not add new forms of computational power.

Artificial intelligence: game search, philosophical issues, neural networks. This is an exploration of one frontier of computational power, incorporating some significant thought about the computational power of the human brain. (Additionally, we can demonstrate that an artificial neural network is computationally as powerful as any similarly connected combination of logic gates.)

Structural linguistics: context-free grammars, regular expressions. These theoretical models tie into the later study of automata, and their relationship in expressive power introduces the concept of computational hierarchy.

Computation theory: finite automata, Turing machines, reductions, halting problem. This final portion relates all of the Turing-complete models, and demonstrates that some problems (i.e., the halting problem) are beyond these models.

For the first half of the course, this outline follows the bottom-up model closely. Thereafter, however, we reinterpret the bottom-up model in terms of developing reductions from one layer to another. This interpretation connects the bottom-up coverage with theoretical computational models, and the overall effect is to draw a broad spectrum of computer science into the theme of computational reductions. By the end, we can draw the picture of Figure 2 uniting most of the course's major topics.

Though this theme allows a broad array of computer science topics, it is important to note that it is not complete. Most significantly, though it would be desirable, the study of computing-related social issues does not fit neatly into this theme. CSCI 150 once included a project in which student groups presented a talk on a topic of their choice relating computing and society. This assignment has finally been eliminated, however, because it has not meshed with the course's theme or approach.

In content, the course matches closely that outlined by Bagert et al., though that paper does not identify such a theme [1]. Two warnings of Bagert et al. match with our own experience: The breadth-first model is difficult to apply

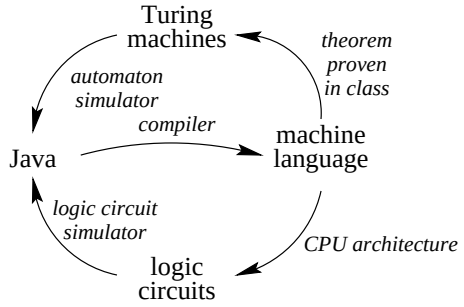


Figure 2: A graph of reductions between computational models.

to students with previous computer programming coursework from high school or other colleges. (Such students are too rare in our case to create an accelerated track [4].) Second, such a course can go too far toward destroying the myth that computer science is about training programmers. (At one point, some of our graduating classes demonstrated low programming competence. While pinpointing the cause is impossible, one possibility is that the low emphasis then given to programming in CSCI 150 led students to consider subsequent coverage as beside the point. A conscious effort to strengthen this unit in CSCI 150 coincided with the emergence of classes with more programming competence.)

Curriculum 2001 adds its own criticism: Breadth-first introductions, it says, can add a full course to the size of the major ([6], p. 31). Our experience does not match with this observation. Students easily transfer the programming unit into their next course, which addresses object-oriented programming and Java more thoroughly. Additionally, they appear to transfer most of the hardware-oriented material into the sophomore-level course on computer systems. Since the coverage of theoretical models appears already to exceed the recommendation of CC2001, further coverage may not be necessary; nonetheless, our curriculum requires an upper-division course including the subject (and we would admit that by that point, students have forgotten much of the automata theory they studied in CSCI 150). We have found, then, that at least 2/3 of the course material transfers neatly into later points in the curriculum. Even if some material requires repetition later, this is consistent with the curriculum design principle of *spiraling* [13].

3. RESOURCES

Over the years, department faculty have developed a wide range of materials for CSCI 130 and 150. Many of the current materials are now available freely through the Web for faculty at other institutions to adapt to their own purposes.

`http://www.cburch.com/socs/`

This section introduces these materials to the wider computer science community.

3.1 Written resources

Though we have examined many published textbooks, none match our selection of material accurately. Thus, we have developed new written resources for students. The CSCI 150 resources include three parts.

- The 100-page textbook is an expanded set of notes developed for the course.
- The 60-page programming supplement describes the fundamentals of Java programming. As a separate component, faculty can easily substitute another introductory programming text (such as Karel [3]), perhaps in another language. The supplement could also be useful in other courses introducing programming.
- The study guide includes a selection of past test questions, with solutions.

3.2 Laboratories and software

Initially, CSCI 150 adopted the schedule suggested by the Denning report: three one-hour lectures and one three-hour laboratory session each week.[2] (In practice, the line between laboratories and lectures is somewhat obscured by frequent in-class exercises during the lecture time, scheduled for a classroom equipped with student computers.) However, because most of the laboratories include a variety of exercises, rather than long projects, we have found little reason for such long laboratory sessions. In Fall 2003, the department changed the course to a schedule of two 85-minute laboratory sessions for every three classes, a schedule that had proven successful with CSCI 130. This modification has lessened student fatigue and reduces the delay between the introduction of concepts in the classroom and the exploration of those concepts in the laboratory.

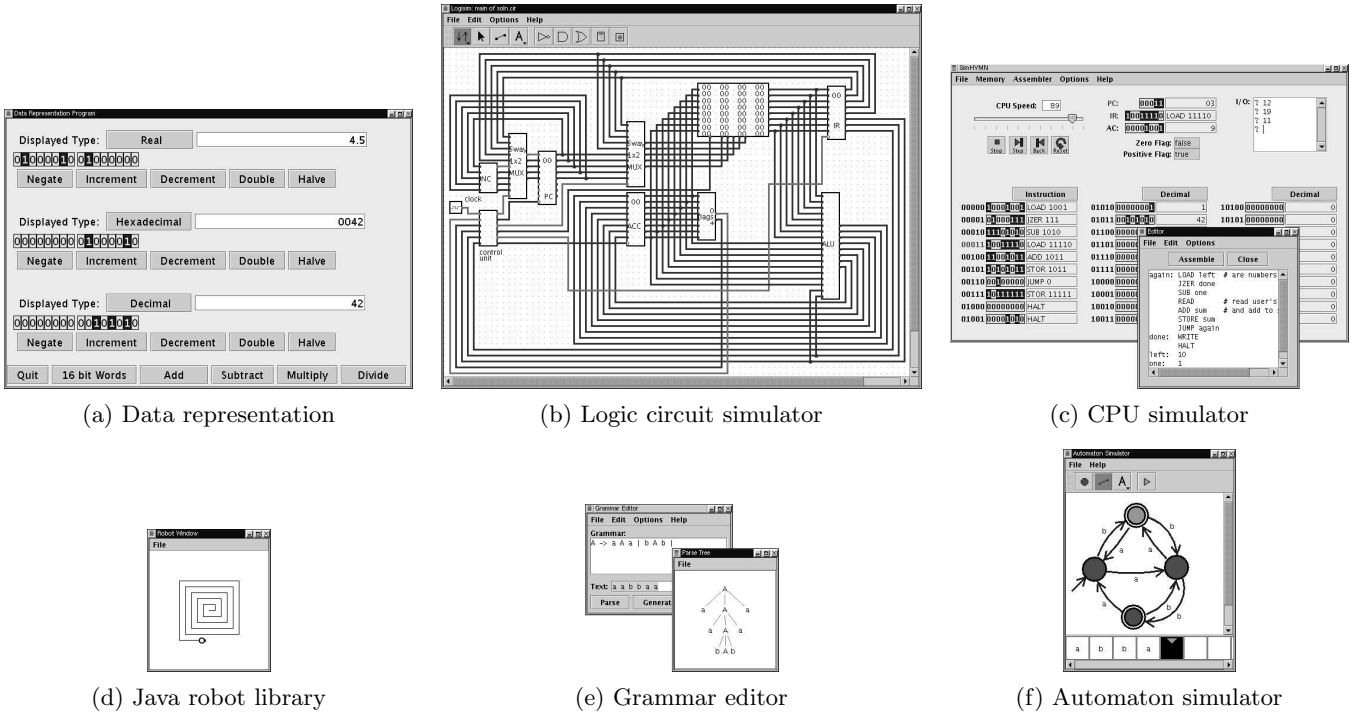
One enticing advantage of the computational-models theme is that it opens the prospect of labs in which students become more familiar with the models by creating designs within their respective frameworks to accomplish assigned tasks. Thus, in one laboratory for circuits, students design a simple circuit for controlling a four-state vending machine for soft drinks. Or when the class studies regular expressions, the laboratory session has students write regular expressions to search a dictionary for, say, words with five consecutive vowels (e.g., *queueing*). Students appreciate this type of laboratory work: In a midterm evaluation form administered in Fall 2003, 29 of 51 students characterized the laboratory assignments as "very useful in learning the material," and another 21 characterized them as "somewhat useful."

This laboratory strategy, though, requires software so that students can work with the models in simulation. To this end, the department has developed a collection of Java applications. All are available to the public via the Web page.

Numeric representation (Figure 3(a)) demonstrates how numbers are represented in two's-complement and floating-point formats. CSCI 150 students spend a laboratory session becoming more familiar with each of these representations.

PNM editor permits students to edit PNM files and to view the images they represent. (The PNM format is a popular image format for uncompressed ASCII images.) CSCI 150 students use this tool as part of one laboratory concerning image representation.

[2]The colleges actually rotate through a six-day cycle. Students attend lectures three days per cycle and a laboratory session once per cycle.



(a) Data representation (b) Logic circuit simulator (c) CPU simulator

(d) Java robot library (e) Grammar editor (f) Automaton simulator

Figure 3: Screen shots of some software resources.

Logic circuit simulator (Figure 3(b)) provides a drawing-program environment in which students can design and experiment with logic circuits [5]. (In early iterations of the course, students used breadboards, but this severely limited the complexity of the circuits they could build. We moved to simulators to enable students to explore deeper concepts.)

In two laboratory sessions, CSCI 150 students use this program to learn about combinational and sequential circuit design. Much later, when they take the sophomore-level computer systems course, they revisit the program and take advantage of its hierarchical design features.

CPU simulator (Figure 3(c)) allows students to experiment with the 8-instruction CPU used in class. The GUI program allows students to view and edit the bits in registers and memory as the computer completes each instruction. The program includes an integrated assembly language editor and a button for stepping backwards through the CPU's execution.

Three CSCI 150 laboratories involve using this simulator. In the first, they write programs by flipping bits (similar to programming the Altair 8800); in subsequent labs, they use the assembler.

The 8-instruction CPU design is simple enough that it can be implemented in the circuit simulator (as illustrated in Figure 3(b)). Although understanding this design does not play a major role in CSCI 150, the instructor can demonstrate the logic circuit as it executes a program. Students in the sophomore-level computer systems course study this design in more detail as part of the unit on architecture.

Java library (Figure 3(d)) includes the robot API (similar to a Logo turtle) used in the Java supplement. CSCI 150 includes five Java laboratories, of which three use this API.

Grammar editor (Figure 3(e)) takes a context-free grammar written in the built-in editor and computes parse trees for text given by the user. This tool plays a role in a CSCI 150 laboratory session concerning context-free grammars and regular expressions.

Automaton simulator (Figure 3(f)) allows students to design and experiment with DFAs, NFAs, DPDAs, and Turing machines in a drawing-program environment. CSCI 150 students use this tool in two laboratories, one for finite automata and one for Turing machines.

Sort animator animates a variety of sorting algorithms. When there is time in the course, students use this program for a simple laboratory assignment concerning sorting algorithms and big-O time complexity.

Lambda calculator demonstrates reductions in the lambda calculus. Although CSCI 150 does not include the topic, the department sometimes teaches the subject in more advanced courses, and this program allows students to experiment with the lambda calculus.

4. CONCLUSION

In our liberal arts context, the introductory survey to computer science, united using the theme of computational models and their relationship, has proven a viable way of structuring the first course in computer science. This theme allows for a broad coverage of the core concepts of computer

science, and it automatically leads to many laboratory assignments for which students experiment with building devices to accomplish tasks within the models they have studied.

We invite other faculty, even where they cannot consider the structure of this particular course, to use the resources introduced in this paper.

5. ACKNOWLEDGMENTS

All of the faculty who have taught CSCI 150 in the past have contributed significantly to its current shape. Besides the authors, these professors include Nathley Caesar, Daniel Challou, Noreen Herzfeld, J Andrew Holey, Chris Lusena, John Miller, and Jim Schnepf.

6. REFERENCES

[1] D. Bagert, W. M. Marcy, and B. A. Calloni. A successful five-year experiment with a breadth-first introductory course. In *Proc. 26th SIGCSE Tech. Symp.*, pages 116–120, March 1995.

[2] D. Baldwin, G. Scragg, and H. Koomen. A three-fold introduction to computer science. In *Proc. 25th SIGCSE Tech. Symp.*, pages 290–294, March 1994.

[3] J. Bergin, M. Stehlik, J. Roberts, and R. Pattis. *Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. 2003. Online [Sep 1, 2003]. Available WWW: http://csis.pace.edu/~bergin/.

[4] K. B. Bruce. Attracting (& keeping) the best and the brightest: An entry-level course for experienced introductory students. In *Proc. 25th SIGCSE Tech. Symp.*, pages 243–247, March 1994.

[5] C. Burch. Logisim: A graphical system for logic circuit design and simulation. *Journal of Educational Resources in Computing*, pages 5–16, March 2002.

[6] *Computing Curricula 2001: Computer Science*. December 2001. Online [Sep 1, 2003]. Available WWW: http://www.acm.org/education/curricula.html.

[7] N. Dale and J. Lewis. *Computer Science Illuminated*. Jones and Bartlett, Sudbury, MA, 2002.

[8] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Communications of the ACM*, pages 9–23, January 1989.

[9] J. Gersting, P. Henderson, P. Machanick, and Y. Patt. Programming early considered harmful. In *Proc. 32nd SIGCSE Tech. Symp.*, pages 402–403, February 2001.

[10] C. Gurwitz. The internet as a motivating theme in a math/computer core course for nonmajors. In *Proc. 29th SIGCSE Tech. Symp.*, pages 68–72, February 1998.

[11] Y. N. Patt and S. J. Patel. *Introduction to Computing Systems*. McGraw-Hill, New York, 2004.

[12] A. Phillips, D. Stevenson, and M. Wick. Implementing cc2001: A breadth-first introductory course for a just-in-time curriculum design. In *Proc. 34th SIGCSE Tech. Symp.*, pages 238–242, February 2003.

[13] K. Powers. Breadth-also: A rationale and implementation. In *Proc. 34th SIGCSE Tech. Symp.*, pages 243–247, February 2003.

[14] C. Shannon. Another breadth-first approach to cs 1 using python. In *Proc. 34th SIGCSE Tech. Symp.*, pages 248–251, February 2003.

[15] S. Vandenberg and M. Wollowski. Introducing computer science using a breadth-first approach and functional programming. In *Proc. 31st SIGCSE Tech. Symp.*, pages 180–184, March 2000.