Experiences and Results from three Years of CSE 211 "Fundamentals of Computing I"

Matthias Scheutz
Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, IN 46556 mscheutz@nd.edu

Abstract – **In this paper we report results from three offerings of CSE211, the first course in a new first-year CSE sequence as part of the new CSE 2002 undergraduate curriculum at Notre Dame (ND02), which was modeled after the suggestions of the IEEE/ACM Computing Curricula 2001. After describing the unique challenges of ND02, we give an overview of ND02 and the role of CSE211 in it. We then summarize the course topics, organization, and infrastructure and present results from formal teacher and course evaluations and student surveys. These results are statistically analyzed to answer among others questions about the utility of open-source tools and programming environments, the utility of SCHEME as a programming language, and the degree to which students' should have prior programming experience in order to perform well in the course.**

Index Terms— first year computer science course, programming environment, SCHEME

INTRODUCTION

During the 2000/01 and 2001/02 academic years, the curriculum committee in the Department of Computer Science and Engineering at Notre Dame re-evaluated its undergraduate computer science and computer engineering curricula. It was decided to adapt both curricula based on the analyses proposed in the IEEE/ACM Computing Curricula 2001 (CC2001), for one to better reflect the changes and demands in computing inside and outside academia, but also to give students more flexibility (in terms of electives) to pursue their particular interests in the various subfields of computing earlier in the curriculum.

The main challenge for formulating computer science and computer engineering curricula at Notre Dame is posed by a set of unique university course requirements, which students have to satisfy at various times during their undergraduate years. In particular, all students in the College of Engineering, which is home to the Department of Computer Science and Engineering, have to take the same courses during their freshmen year, effectively leaving only three years for courses in their various majors. Consequently, none of the model curricula proposed by CC2001 could not be adopted without adaptation. In particular, it became necessary to develop a new *First-Year Computer Science and Engineering Sequence CSE211/212*, which is specifically targeted at a 3-year CS/CE curriculum and thus focuses on providing a more comprehensive overview of computer science together with more opportunities to acquire problem solving and programming skills as would otherwise have been necessary within one year.

In this paper, we describe our experiences with three offerings of the first course in this sequence, *Fundamentals of Computing I* (CSE211), and present results from statistical analyses of student performance based on formal teaching evaluations, student surveys, and course grades. Specifically, we look at (1) the relation between prior programming experience and overall performance in the course, (2) the utility of using different kinds of open-source tools (in this case SCHEME compilers, editors, and integrated development environments), and (3) the students' perception of the utility of SCHEME as a programming language for this course.

MEETING THE CHALLENGES OF A 3-YEAR CSE CURRICULUM–THE DESIGN OF CSE211

Engineering undergraduates at Notre Dame have to learn the subject material of their major in three, rather than four years. The reason for this time restriction is that the first year engineering students take general education courses required by the university and a mandatory one-year general introductory engineering sequence EG111/112 that touches on aspects in various subfields of engineering. As a consequence, core CS/CE materials that are usually spread over two years at other institutions have to be covered as quickly as possible (ideally in two, but at most in three semesters). For the rest of the paper, we concentrate on the first introductory course in computer science *CSE 211 Fundamentals of Computing I*.

The design of CSE211 departed from that of a previously offered course CSE233 *Functional Programming*, which was essentially based on MIT's 6.001 *Structure and Interpretation of Computer Programs*. It was decided to use Abelson and Sussman's equally named book (SICP) [2] for CSE211, given it's educational success and comprehensive, conceptual approach to computing and problem solving. Particularly appealing was the book's quick focus on procedural and data abstraction and its methodological goal of keeping new syntactic constructs to a minimum, which is facilitated by using SCHEME as a programming language. Given that about half of the incoming CS and CE students typically have only very limited programming backgrounds, it was believed that a syntactically simple language like SCHEME would be easy and quick to learn, and more importantly, allow students to focus on programming concepts right away (rather than having to spend a significant time on learning syntactic constructs as

0-7803-9077-6/05/$20.00 © 2005 IEEE

is the case with syntactically complex languages like C++).[1]

I. Lecture Topics in CSE211

The following lecture topics for CSE211 were determined as part of an overall assignment process that mapped the core coverage as specified by CC2001 onto core courses in ND02:

Procedural abstraction: : simple functions; parameters and results; composition; conditional expressions

Recursion: : the concept of recursion; recursive specification of mathematical functions (including inductive definitions); simple recursive procedures, mathematical functions (such as factorial and Fibonacci); simple recursive procedures (Towers of Hanoi, permutations, fractal patterns); implementation of recursions

Data abstraction:: list structures; hierarchical data; symbolic data; the importance of data abstraction

Algorithms and problem-solving:: problem-solving strategies; the role of algorithms in the problem-solving process; implementation strategies for algorithms; debugging strategies; the concept and properties of algorithms (brute-force algorithms; greedy algorithms; divide-and-conquer; backtracking; numerical approximation algorithms

Object-oriented paradigm:: object-oriented design; encapsulation and information-hiding; separation of behavior and implementation; classes, subclasses, and inheritance; polymorphism; class hierarchies; collection classes and iteration protocols; fundamental design patterns

Basic computability theory:: tractable and intractable problems; the existence of noncomputable functions

Basic computational complexity:: asymptotic analysis of upper and average complexity bounds; big-O notation; standard complexity classes; empirical measurements of performance

Overview of programming languages:: history of programming languages; brief survey of programming paradigms; the role of language translation in the programming process

Fundamental programming constructs:: syntax and semantics of a higher-level language; variables, types, expressions, and assignment; simple I/O; conditional and iterative control structures; functions and parameter passing; structured decomposition

Evaluation strategies:: representing computation state; streams; lazy evaluation; nondeterminism; the construction of an interpreter

Software development methodology:: Fundamental design concepts and principles; structured design; testing and debugging strategies; test-case design; programming environments; testing and debugging tools

[1]It should be noted that by taking the mandatory introductory engineering sequence EG111/112, even students without any prior programming experience will get some exposure to programming. The following excerpt is from the EG111/112 course description: "The goal in EG 111/112 is to get first-year students to 'think like a programmer', a goal that is not language-dependent. In EG 111/112 you will learn the basic constructs that can be found in any computer language 'mechanisms such as primitive expressions, variables, functions, and programs.' You will use MATLAB and C and Not-Quite-C to write software using a top-down-design/bottom-up-implementation strategy that is applicable no matter what kind of computer you are using or what language you are programming in."

TABLE I
THE TOPICS COVERED IN CSE211 WITH LECTURE HOURS SPENT ON THE TOPIC IN PARENTHESES.

Time spent on Topics in CSE211
Graphs and trees (3)
Fundamental programming constructs (3)
Algorithms and problem-solving (2)
Fundamental data structures (6)
Recursion (5)
Basic algorithmic analysis (2)
Algorithmic strategies (2)
Fundamental computing algorithms (4)
Basic computability (1)
Overview of programming languages (1)
Declarations and types (1)
Abstraction mechanisms (2)
Functional programming (4)
Concurrency (2)
Software design (1)
Software tools and environments (1)
History of computing (1)

Machine level representation of data:: bits, bytes, and words; numeric data representation and number bases; signed and twos-complement representations; representation of non-numeric data

Table I gives a summary of the topics and the respective numbers of lectures spent on them out of 41 lectures for a 3 credit hour course. These topics can roughly be mapped onto SICP chapters 1 through 4: "Ch.1: Building Abstractions with Procedures" (9 lect.), "Ch.2: Building Abstractions with Data" (9 lect.), 'Ch.3: Modularity, Objects, and State" (13 lect.), and "Ch.4: Metalinguistic Abstraction" (10 lect.). Even though SICP is very comprehensive (especially for a beginning course text), it does not sufficiently cover all the required topics for CSE211, in particular, object-oriented design and programming as well as basic principles of parallel programming and process synchronization for which additional lecture notes and handouts were prepared. Also, topics in the history and theory of computation, an overview of programming languages, software design principles, machine representation, and other topics are either not covered at all or not in enough detail in SICP. At the same time, ch. 5 in SICP deals with topics of compilers, which were found to be outside of the scope of the first programming course.

II. Course Organization

CSE211 consists of a three-credit hour lecture with a mandatory one-credit hour laboratory section, in which students can practice and elaborate the concepts presented in the lecture. Grades are given only for the lecture part and can be broken down into 50% for assignments (25% for weekly individual and 25% for five large group assignments), 5% for class participation and attendance (which is mandatory in both sections), 15% for three short examinations after the first three chapters in SICP (5% each), and 30% for the comprehensive final exam (to ensure that students are capable of mastering the whole

course content).[2]

Individual assignments are intended to rehearse concepts presented in the lecture to ensure that students keep up with the class material. Group assignments are intended to cover a major course topic and enforce cooperative learning. They not only further develop concepts from the lecture, but also introduce new major topics in computer science and allow students to develop the communicative skills necessary for future collaborative work (e.g., in industry or academia) by giving them the opportunity to discuss and develop ideas together with their peers to utilize the "synergy effect" that is not possible with individual assignments.

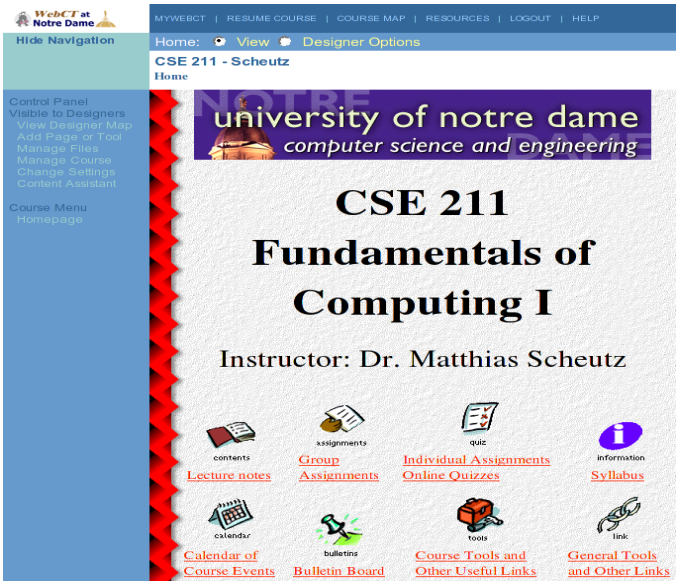

FIGURE 1.
THE TOOLS IN WEBCT COURSE HOME PAGE.

To reduce the bookkeeping efforts (for students and the instructor alike), the online WebCT [3] teaching tool was used to organize the various aspects of the course (see the snapshot of the main WebCT page in Figure -II). In this environment, students have easy access to basic course materials such as the syllabus, lecture and lab notes together with supporting code (which are provided in addition to the book), individual assignments (which can and are partly be auto-graded to save grading effort), and group assignments (including supporting files). In addition, they can access various online resource (e.g., SICP and its supporting code, SCHEME manuals, etc.), view their current standing in the course (i.e., their fraction of the percentage possible at any given point), use the course calendar with notification about reading assignments and upcoming deadlines (e.g., for assignments or quizzes), post questions to one of several bulletin boards (for the lecture, the lab, the assignments, the programming environment, etc.) and ideally answer question of their peers (which count towards class participation).

Finally, it is worth mentioning that part of the intent of the ND02 was to require students to take CSE210 (*Discrete Mathematics*) as early as possible in the curriculum to provide them with a good formal, theoretical foundation. For this reason, CSE210 has to be taken in the third semester, the same as CSE211. Hence, it was only natural to synchronize the two courses in the sense that overlapping topics (such as recursion and inductive definitions, complexity of programs and big-O, first-order logic and reasoning, search algorithms, theory of computation, number representations, and several others) be introduced the same time (to the extent possible), which makes it possible to assign programming exercises in CSE211 to reinforce concepts from CSE210, which frees up exercise space for more theoretical questions (rather than programming assignments).

III. Group Projects

The group projects were intended to allow students to collaborate in teams in order to strengthen their understanding of the lecture materials. They require students to read, understand and solve a novel problem that they had not encountered in the lecture before. Group project typically integrate several lecture topics and are designed to be too comprehensive to be solved by individual students in the given time. The topics of the projects were:

Recursion and basic functions: Students had to compute sequences of numbers using recursion, and fill in function templates to complete computations.

Recursion and data structures, multiple representations of data types: In the first part, students had to write various basic functions for RSA encryption and decryption, in the second they used coercion and methods for representing multiple data types to perform operations on polynomials and function spaces.

Pattern matching and tree search: Students had to implement a production system that uses pattern matching to match right-hand sides of rules against facts in a knowledge base to instantiate left-hand sides for rules as new goals for recursive backward chaining until all goals are satisfied (i.e., matched by facts).

Object-oriented programming: Students had to implement new methods and new objects in an object-oriented version of SCHEME (both with and without special syntax support for objects).

Program interpretation and register machines: Students had the option of either implementing code for simulated register machines in a simplified assembly language or to write a small interpreter for their favorite programming language (those projects typically used a subset of BASIC).

IV. Computing Infrastructure

Two different open-source, cross-platform computing environments were employed. In 2002, the KAWA [4] SCHEME interpreter for the JAVA virtual machine was used together with XEmacs [5] as integrated development environment. The

[2]In the 2004 offering, the final exam was reduced to %25, while the individual assignments were increased to 30% based on student feedback.

use of KAWA (and not any of the other available SCHEME interpreters) was preferred to allow for the introduction of JAVA in the second half of CSE211 in anticipation of JAVA being the language for CSE212. In particular, the idea was to start with JAVA syntax as part of the lectures on object-oriented design and allow students in some assignments to use mixed programming language use (e.g., direct access to the JAVA graphics API within SCHEME as is possible in KAWA or mixed code source files for which we created a pre-processor to work with the SUN JAVA compiler). This plan, although originally envisaged, was however abandoned during the Fall 2002 semester for practical reasons in favor of using C++ in CSE212. The subsequent two course offerings in 2003 and 2004 then more mature and easier-to-use integrated DrScheme environment [6] instead.

The labs as well as individual and group assignment required students to turn in SCHEME code. Both computing environments thus had to be cross-platform, because the lab sessions were held in a room with SOLARIS machines, while most students used Windows clusters on campus or their own Windows machines at home for assignments (in 2004, additional Linux clusters with RedHat Linux were available and used by students). In this respect, DrScheme was much easier to install, maintain, and operate than the KAWA/XEmacs combination, which requires more knowledge about the underlying operation system to work properly.

RESULTS FROM 3 YEARS OF CSE211

Three main hypotheses underwrite the design of CSE211: (1) it is feasible to include materials in the first introductory course that are intended for the second according [1] without sacrificing the students' level of understanding of other materials, (2) using SCHEME as a programming language eliminates advantages and/or disadvantages some students might have based on their high school programming background, and (3) an appropriate programming environment in combination with a syntactically simple programming language is critical to students' learning and perception of the course.

The three hypotheses were tested in three subsequent offerings of the course. The data available for analysis are the formal teacher and course evaluations administered and analyzed by the Office of Institutional Research at Notre Dame, student surveys conducted at the end of the first two course offerings and at the beginning of the third course, and student performance in the course as measured in terms of their grades on the various components (assignments, quizzes, final, etc.). The results in Table II show averages for the three offerings for each category/questions as well as p-values of T-tests (t-values are omitted for lack of space) comparing them in consecutive years (i.e., 2002 vs. 2003, and 2003 vs. 2004–bold-faced entries indicate significant differences for $\alpha = 0.05$). The ratings were 4=*strongly agree*, 3.2=*agree*, 2.4=*indifferent*, 1.6=*disagree*, 0.8=*strongly disagree* in the student survey and 4=*excellent*, 3.2=*good*, 2.4=*satisfactory*, 1.6=*poor*, 0.8=*very poor* for the rest.

TABLE II

RESULTS FROM TCES (TOP) AND STUDENT SURVEYS (BOTTOM) IN THE THREE CSE211 OFFERINGS (SEE TEXT FOR DETAIL).

Overall Student Perception	'02 N=28	p-val	'03 N=30	p-val	'04 N=38
Perception of Teaching	3.2	.148	3.5	.160	3.3
Course Content	**2.5**	**.049**	**2.9**	.497	2.9
Aspects of Skill Level.					
Rationality/Problem Solving	3.0	.114	3.1	.427	3.2
Skill Development	3.1	.276	3.5	.081	3.3
Factual Knowledge	–	–	2.2	.828	2.4
Developing Creativity	3.2	–	–	–	–

Student Survey	'03	'02
SICP was appropriate for this course	3.1	3.1
Lecture notes helpful in addition to SICP	3.6	3.6
The course organization in WebCT was useful	3.3	3.2
Split into individual and group assignments was useful	3.1	2.8
Individual assignment helped understand lecture better	**3.1**	**2.5**
Group assignments helped to understand lecture better	3.0	3.1
The SCHEME implementation worked well for me	**2.9**	**2.4**
The programming environment worked well for me	**3.2**	**2.5**
Viewing my current standing in WebCT was useful	3.1	3.3
Getting feedback in WebCT was useful	3.3	3.3
Summary of instructional goals was useful	3.5	3.4
Exams tested exactly the instructional goals	3.0	2.8
Discussion of exam solutions was helpful	3.0	2.9
Exams tested my knowledge of course material thoroughly	2.9	2.8
I feel I have a good overview of different aspects of CS	3.1	2.9
Have good idea of CS topics coming up later in ND02 Curr.	3.1	2.7
I feel my programming skills improved significantly	3.2	2.9
I see the utility of SCHEME as an instructional language	2.8	2.3
I learned to decompose complex problems into simpler ones	3.1	2.9

The questions we are most interested in pertain to overall quality of teaching, the student's perception of appropriateness of course content, the employed programming language and environment, and the degree to which prior programming experience has an effect on performance in the course.

Overall, there was no difference among the three offerings with respect to students' perception of overall teaching quality, which they rated as very good on average (although there were interesting significant differences in the various components of teaching, which we cannot address here for space reasons). This is important, for the quality of teaching and the interaction of faculty with students is critical to learning [8], and also to keep attrition rates in first-year courses low [7] (compared to national trends in the US, the attrition rates in CSE211 are very low: 5/36 in 2002, 3/40 in 2003, and 4/48 in 2004).

Moreover, there was no significant difference in students' perception of what they learned in CSE211. Most students thought that they learned how to solve problems well (16/28 in 2002, 20/30 in 2003, and 25/38 in 2004), followed by very good skill development (11/28 in 2002, 5/30 in 2003, and 8/38 in 2004), both in line with the course objectives.

Most differences in student perceptions occurred between the first and the second offering. The significant difference in perception of the appropriateness of the course content is particularly striking given that the whole organization and

content of CSE211 was *identical* between both offerings (i.e., the course materials were identical including the schedule, reading assignments, timetables, WebCT environment, etc.; moreover individual and group assignments, quizzes, and exams were very similar). Given that the class size was also about the same, these results suggest that the difference in perception with respect to the course content might be due to the difference in programming environments. This surprising conclusion is supported by the highly significant differences in student satisfaction with the particular implementation of the programming language (KAWA vs. MzScheme, p=.005) and the supporting programming environment (KAWA/XEmacs vs. DrScheme, p=.004).[3] Further evidence comes from the significant difference in the perception of the utility of individual assignments to understand the lecture material better (p=.038), which benefit most from an "easy-to-use" programming environment like DrScheme (which allows students to check their answers quickly, while more complex environments like KAWA/XEmacs act rather as a deterrent–this was also confirmed by written comments on TCEs). This is also evident from the students' perceptions about how much time they spent on average on the course (3.64/4 in 2002, 3.23/4 in 2003, and 3.45/4 in 2004, where 4=*much more than average*, 3=*more than average*, 2=*average*, and *1=less than average*, the difference between 2002 and 2003 is significant, p=.02).

One of the consequences might have been the trend towards a statistically significant difference in students' perceptions of the utility of SCHEME as an instructional language (p=.099). While students in both years perceive SCHEME as somewhat useful, the 2002 students are leaning on average more towards being indifferent on the question while students in 2003 tend to agree with it. Interestingly, despite all the above differences, there is no significant difference between students' performance on the finals (64.6/100 in 2002 and 64.8/100 in 2003). Moreover, there is no significant difference among the final grades and grade distributions in all three course offerings.

Other difference between course offerings are related to prior programming experience. While in 2002 and 2003 about 50% (15 out of 31 and 19 out of 37) had prior exposure to C++, only 40% (18 out of 45) in 2004 had C++ experience (and 25/48 had no programming experience whatsoever). None of the students in all three offerings had prior experience with SCHEME. Correlating prior programming experience and final grade (here we have data only for 2004), however, gives $r = .11$, while correlating C++ and final grade yields $r = .18$, showing that in both cases programming experience is not a factor in overall course performance–this is in line with the findings by Brockman (oral communication) in the context of a design project [9].

[3]We believe that KAWA lacks the level of maturity of DrScheme; throughout the semester, we witnessed various problems with KAWA and a few times we even discovered bugs in the implementation that required us to find workarounds. Moreover, some understanding of the underlying operating system is needed for effective work with the KAWA/XEmacs combination, while DrScheme is largely self-contained and can be used without such expertise.

CONCLUSION

The results from three offerings of CSE211 show that introductory topics in CS can be feasibly covered in the first course without sacrificing the students' level of understanding and that a programming language like SCHEME, in which none of the students had prior experience, eliminates advantages and/or disadvantages of prior programming experience (or lack thereof). We found no indication that prior programming experience (in C++ or other languages) has any impact on student performance in the course. Moreover, students on average tended to see some utility of SCHEME as an instructional language. This is particularly important in the light of pressure from industry to move into C++ quickly.

Finally, our analyses suggest that the right programming environment is a critical component in a first-year CS course. Specifically, as in the case of CSE211, the choice can contribute to students' perception of the overall course content and lead to significantly higher time requirements. However, in CSE211 it did not have an impact of overall student performance. Hence, open-source tools are appropriate programming environments as long as they have reached a high level of maturity and are easy to use (like DrScheme).

Overall, we believe that CSE211, while clearly being difficult and time-consuming for students, is a viable first course in CS that achieves its instructional objectives and requirements in the context of the ND02 curriculum. Naturally, there is still much for improvement. Of particular concern are currently the high time requirements for students.

Future versions of the course will address this problem by experimenting with different distributions of individual and group assignments, possibly reducing the number of assignments and quizzes in favor of more interactive practice in the lab and the lecture.

ACKNOWLEDGMENT

The author would like to thank the members of the CSE undergraduate curriculum development committee 2000-02 (headed by Jay Brockman) in the department of computer science at Notre Dame for their hard work in defining ND02 and the department for having been given the opportunity to develop CSE211 and teach it in three consecutive years.

REFERENCES

[1] The IEEE/ACM Computing Curricula 2001 (CC2001). http://www.computer.org/education/cc2001/
[2] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed., McGraw-Hill, 1996.
[3] WebCT. http://www.webct.com/
[4] KAWA. http://www.gnu.org/software/kawa/
[5] XEmacs. http://www.xemacs.org/
[6] DrScheme. http://www.drscheme.org/
[7] J.M. Cohoon and L.Y. Chen, "Migrating Out of Computer Science", *Computing Research News*, Vol. 15/No. 2, pp. 2-3, 2003.
[8] A. Linse, W. Jacobson, L. Reddick, "Toward the Best in the Academy", Essays on Teaching Excellence Vol. 16, No. 7, 2004-2005.
[9] J.B. Brockman, "Evaluation of student design processes", *Frontiers in Education Conference*, 1, 6-9, pp. 189 - 193 vol.1, 1996