What Comes After CS 1 + 2: A Deep Breadth Before Specializing

Troy Vasiga
Department of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1 Canada
tmjvasiga@math.uwaterloo.ca

Abstract

There has been much discussion of CS1 and CS2 in computer science education circles. This paper presents a proposal for a course subsequent to CS2 that acts as a "springboard" for students diving into more specialized Computer Science courses at the upper year levels.

1 Introduction and Motivation

To say there has been much discussion of CS1 and CS2 in SIGCSE circles is a gross understatement. To quantify this statement, consider that from the 78 papers accepted to SIGCSE 2000, at least 18 of them (23%) were focussed on CS1 and/or CS2 [12]. The motivating question underlying this area of research seems to be: "What are the fundamental elementary CS concepts that need to be conveyed to novice CS students?"

Additionally, there is a focus in the computer science teaching community concerning teaching specialization courses, such as operating systems, concurrent programming, etc. That is, these specialized courses expect students to begin the course with "enough" underlying computer knowledge to avoid having to review fundamentals to any great length. Moreover, since there are a wide range of specialization courses, the following question seems natural to ask: "What are the fundamental pieces of knowledge required for specialization courses and how can this knowledge be expressed succinctly to students?"

These two questions beg the connecting question: "What might glue or lead the first-year courses into the specialization courses?" In this paper, we present an outline of such a course, CS 241 [11], that follows naturally from CS1 and CS2 and presents a context for specialization courses in upper years. It should be noted that while the specific course CS 241 exists, this paper will attempt to convey the underlying educational methodologies of CS 241 to be of benefit to other instructors or curriculum developers.

2 Overview of the Course

The course CS 241, Foundations of Sequential Programs, has been offered for many years at the University of Waterloo. The current structure of the course has crystallized (to some degree) during the 2000 academic year. The course is taken by computer science major students in their third academic term, and enrollment ranges between 100-400 students per term.

The general sequencing of the course is outlined below:

- Assembly language: In this section of the course, we introduce the basic DLX assembly language (as outlined by Patterson & Hennessey [7]), presenting the basic concepts of the parameter stack, registers, recursive calls and program counter. Students write several assembly language programs, including programs that use subroutines, programs that are recursive, and programs that access an array in RAM.
- Regular languages: This section covers regular expressions and finite state machines (in all their forms) in their relation to scanning. Students construct many regular expressions for various languages, and implement a scanner (using both finite state machines and regular expressions) for a simplified language (SL). SL is a subset of C or Pascal (this varies from term to term), which includes elements such as conditionals, repetition, variables and pointers (no subprograms, objects or arrays).
- Context-free languages: The material covered in this section covers basic grammar structure, ambiguity,

LL and LR grammars, both within and outside the application of parsing. Students construct a context-free grammar that formally defines SL programs.

- Attribute grammars: This section covers the addition of computation rules to a given grammar to gain "more power" and to do actual translation from one language to another. This section ties back into the Assembly Language section outlined above by taking a high-level language and demonstrating how DLX code can be generated by way of attribute grammars. Students add attribute computation rules to generate DLX machine code equivalent to a given SL program.
- Assemblers, Linkers, Loaders: This section delves deeper into the low-level instructions that occur when an assembled program is executed. Students (in some offerings of the course) implement an assembler, using the techniques of scanning, parsing and attribute computation presented in the earlier sections of the course.
- Scheme: This section acts as a foil to illustrate the basic differences between procedural and functional programming languages. Students write curried and nested functions to accomplish a variety of processing tasks, including implementing a symbol table by way of a binary dictionary.

The languages used by the students in the course are a home-grown DLX assembler and interpreter, Java and Scheme. Using these languages, students extract the theoretical concepts taught in lectures (3 hours per week) to construct a compiler for SL.

Additionally, students use other compiler construction tools, such as JLex and CUP, which are outlined in Appel [4] and available on-line [3].

In the remainder of this paper, we present the rationale for using this course as a springboard after CS1 and CS2 and before students enter specialized CS courses.

3 The Need for CS3

In this section, we examine the fundamental concepts that students should attain by the end of CS1 and CS2. We then make the case that these skills are not ideal (on their own) for delving into specialization courses.

We begin by outlining the basic assumptions for objectives taught in CS1 and CS2, based on courses currently offered at various institutions, as well as the ACM Computing Curricula 1991 [1].

3.1 Basic Assumptions About CS1 and CS2

By the end of CS1, students will have an understanding of the following concepts, both on a theoretical and constructive level:

- algorithms and their usage in problem solving
- basic programming structures, such as condition evaluation and repetition
- basic data types (including integers, strings, and arrays/vectors)
- object-oriented concepts such as objects, classes, and inheritance

The above concepts are those covered by most CS1 courses (such as those listed at [5, 6], to cite two of many), and are the common chapters in introductory computer texts, such as Horstmann [8].

By the end of CS2, it is assumed that students will have the following knowledge, both from an applied and theoretical perspective:

- more advanced data structures (stacks, queues, trees)
- abstract data types
- recursion
- sorting (including quicksort and mergesort)

The above concepts are covered in many computer science courses, including [9, 2].

3.2 The Case Against Specialization in CS3

Given the above assumptions about the educational outcomes of CS1 and CS2, we now make the case that students are not fully prepared to specialize into areas such as software engineering, operating systems, hardware, theory or concurrent programming immediately following CS2.

There are three pedagogical reasons for not following CS2 with the specialization courses:

1. **Specialization early may cause a lack of synthesis.** We consider this point in two cases: in theoretical courses and in applicative courses.

   In theoretical courses (which would, arguably, include hardware and "proper" theory), the student would have no context nor reason to relate various theoretical constructions, numerical representations, truth tables, gates and other circuits. These concepts will remain detached from the students schema unless concrete connections are made with subjects students are already familiar with. After CS2, students may only have (at best) a vague notion of how binary representations (in a hardware sense) or finite state machines (in both a hardware and theoretical sense) relate to the computational essence of computers.

In applicative courses (including operating systems and concurrency), students will not have been exposed to the interaction between the execution of a (high-level) program and the underlying CPU structure. This concept is crucial in order to place the idea of concurrency and parallelization in context, since concurrent programming concerns itself with managing this interaction. In terms of operating system courses, the various levels of the OSI model (see [10]) become clearer in the context of differentiating the high-level application level from the low-level data level.

2. **Students may be ill-prepared in terms of the size, complexity and structure of programs that they need to comprehend and create.** In general, students in CS1 and CS2 courses are asked to create parts of larger programs, or a collection of smaller programs. In other first-year courses, students work on a course-long project, adding functionality as the term progresses. A prime example of this would be a strategic game/simulation where various features are added and improved on as the course progresses, with each new feature introducing a new concept.

   The difficulty with this pedagogical model is that it doesn't scale well when larger problems need to be solved. In particular, compilation is most easily viewed as a *sequence of information processing steps*, which is a fundamental concept when dealing with operating systems (in the OSI model stated earlier). As an example from a specialization course, the problem of creating an operating system is not to figure out what functionality needs to be included, but how to process information into the desired outcome.

   To summarize this point, the specialization courses require more problem solving tools and techniques than those taught in CS1 and CS2.

3. **Students may only view the computer as an algorithm entry device, and not view it as a system.** Students who complete CS2 tend to be proficient programmers, and are usually adept at taking a well-formed specification of a problem and creating an outline of how they would solve it algorithmically. However, specialization courses require a deeper understanding of the physical machine and its functionality, rather than a simple view of how to write a program. Students who have completed CS2 do not have this necessary global view of the computer system.

3.3 The Case Against Depth or Breadth in CS0

An alternative approach (that some readers have suggested) to this CS3 course could be to add breadth to CS0, in the form of presenting a global view of the compilation process. This alternative idea will be refuted in this section.

CS0 is a course which focuses on taking students from a near-zero level of CS knowledge (both in the programming aspect and in overall computer comfortability) to a level of basic programming skills and general computer usage ability. From personal experience, this very simple goal is not always achieved in CS0 courses: to conceive of adding more material and goals is unreasonable. Moreover, to expect students to progress from no programming experience to a level of programming knowledge high enough to understand how programs can be used to translate other programs into binary code is asking too much of CS0. The CS0 course works best when the basic usage elements of computers are the focus, not the advanced implementation details.

4 The Case for CS 241 as CS3

In this section, we make the case for having a course similar to CS 241 as the springboard CS3 course before students take specialization courses.

4.1 Looking Back

We now make the case that CS 241 follows in a continuous way from CS2. We examine issues of modularity, abstraction and data types.

Modularity The concept of modularity is incorporated into CS 241 by way of constructing a compiler in modular components that are sequentially connected together. That is, students first construct a scanner, then a parser, then an attribute-computation tool to finally output assembly language. This structure of program design expands on the basic modularization techniques of incremental design by way of introducing component-based modularization with increasingly complex functional components which concatenate together to produce one system.

Abstraction The concept of abstraction is extended upon in CS 241. In particular, CS 241 exposes several layers of abstraction that most students didn't even know existed: the layers between a high-level programming language and the actual CPU. Additionally, the benefits of abstraction are clearly highlighted, by way of illustrating that abstracting away the binary representation of instructions is a *good thing*, since for the most part, programmers do not want to be concerned with this low level. To put this another way, if there was a slogan for this proposed CS3 course, it is that we attempt to *remove the mystery of how programs work.*

Data Types In CS 241, students apply their knowledge of stacks (for assembly language programming and parsing), trees (for representing parse trees), and lists (for Scheme representations of data).

4.2 Looking Forward

In this section, we outline how CS 241 leads elegantly into the specialization courses offered in upper years. We look at each specialization course in turn.

Hardware CS 241 leads students down to the machine language level: that is, after compiling and assembling, students see how a high-level language is equivalent to binary machine code. The very natural question that follows is to ask "How does the machine use machine code?" This question is the focus of computer architecture and design courses.

Theory Based on the introduction of the practical side of finite state machines and context-free grammars, students should formalize these notions. In particular, learning how to determine if a language is regular (or not), context-free (or not) provides a clearer understand of the (relative) power of regular expressions and context-free grammars. Thus, the practical introduction of formal languages in a compilation sense motivates further study in a deeper theoretical sense.

Computer Graphics and Real Time Since graphics and real time programming are concerned with speed and efficiency (to a large degree), the connection between high level languages and individual machine instructions becomes crucial. In particular, the ability to understand how many CPU clock cycles an instruction takes, whether it is a high-level, assembly-level or machine-level instruction, is a key (if not central) component of both these courses.

Artificial Intelligence In CS 241, we illustrate the search technique of backtracking in the context of trying to parse a string (i.e., determine whether a given word w is in a language defined by some context-free grammar). This idea of backtracking forms the essential core of artificial intelligence, in terms of searching "intelligently" as opposed to searching "brutally."

Software Engineering Software engineering requires software as its "material." Moreover, the usefulness of software engineering is directly proportional to the size of the program under discussion. At the end of CS 241, students have a compiler for a simple language as an artifact. A first exercise in a subsequent software engineering course could be: "Change your compiler to now work for a modified language and output CISC assembly code." The concepts of documentation, consistent pre/post conditions, and modularity would become immediately obvious to the student while working through this exercise.

Compilers and Programming Languages CS 241 prepares students for compilers and programming languages by exposing them to the essential decisions that go into constructing a compiler. For instance, students learn how ambiguity in a context-free grammar which specifies a high-level language can result in extremely different and unpredictable functionality.

Additionally, since students are exposed to a functional programming language, the concept that not all programming languages need to follow a procedural paradigm is emphasized, opening the door for more "interesting" languages like ML, APL, and Prolog.

5 Conclusions and Further Research

In concluding this paper, it should be noted that this course outline may not apply to every school setting. However, the point of this paper is two-fold: curriculum changes are easier when shared amongst schools, and this sharing can occur only through reasoned discussion. In some senses, this paper can be considered a "first shot over the bow" for opening post-CS2, pre-specialized curriculum discussion. As such, the author very much welcomes views of how other institutions present this material, or why they choose not to, and what ramifications it has on curriculum.

In terms of further research, the author is considering polling upper year students to see how their knowledge gained in CS 241 has been applied in upper year courses. As well, the possibility of pre-testing/post-testing students on their knowledge could also be explored.

As a final point, it is worth noting that many students report that at the end of this course "the whole thing tied together" and "programming now makes sense" on their course evaluations. This lends more evidence to the benefit of this course.

References

[1] ACM Curriculum Committee on Computer Science. *Computing Curricula 1991, Report of the ACM/IEEE-CS Joint Curriculum Task Force* (1991), ACM Press.

[2] Albrech, D., and Pickett, D. Courseware for cs1303, 2001. Online. Internet. Available WWW: http://www.csse.monash.edu.au/courseware/cse1303/

[3] Appel, A. W. Modern compiler implementation, 1998. Online. Internet. Available WWW: http://www.cs.princeton.edu/~appel/modern/java/.

[4] Appel, A. W. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[5] Becker, B. W. CS 130 course outline, 2001. Online. Internet. Available WWW: http://www.student.math.uwaterloo.ca/~cs130.

[6] Covington, R. Caltech CS2, 1998. Online. Internet. Available WWW: http://www.ugcs.caltech.edu/ cs2/lectures/010300/.

[7] Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, 1990.

[8] Horstmann, C. *Computing Concepts With Java Essentials.* John Wiley & Sons Inc., 1999.

[9] Pretti, J. P. CS 134 timetable, 2001. Online. Internet. Available WWW: http://www.student.math.uwaterloo.ca/~cs134.

[10] Tanenbaum, A. S. *Modern Operating Systems.* Prentice-Hall, 1992.

[11] Vasiga, T. M. J. CS 241, 2001. Online. Internet. Available WWW: http://www.student.math.uwaterloo.ca/~cs241.

[12] Walker, H. Letter from the program chair/table of contents. *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education* (2000), iv–xix.