Introductory Computer Science: The Case for a Unified View

J. Stanley Warford

Computer Science Department
Pepperdine University
Malibu, CA 90265

Introduction

A recent paper by Gibbs and Tucker [1] pointed out the desirability of new directions in our thinking about the computer science curriculum. They observe that the

> ... core curriculum in computer science is frequently questioned because it seems to be composed of a collection of different programming and applications courses and fails to explicate adequately the principles that underlie the discipline.

Their goal is to incorporate into the curriculum a view of computer science "as a coherent body of scientific principles ... rather than allow it to be driven by the needs and priorities of particular technologies".

The first purpose of this paper is to affirm that Gibbs and Tucker are correct in their observations. A major shortcoming of the typical introductory course is its failure to introduce students to computer science.

The second purpose is to propose an introductory course structure that helps to achieve the noble goal of teaching the true fundamentals of our discipline. This paper proposes that the best framework in which to present a unified view is the concept of levels of abstraction.

The next section presents the concept of levels of abstraction as a unified view in the introductory computer science course. The following section then presents an implementation of the concept. The implementation has three components: an introductory computer science course sequence taught at Pepperdine University, a machine designed to teach the fundamentals of classical von Neumann architecture, and a textbook in preparation based on the course.

© 1988 ACM 0-89791-256-X/88/0002/0044 $1.50

Concept

The proposed model for the introductory computer science course is best illustrated by the typical course in physics, a much older discipline. Like computer science, physics is a broad area. The main areas of study include mechanics, thermodynamics, electromagnetics, and modern physics.

Over the years a traditional introductory physics course has evolved that is uniform in content from school to school. Physics educators realize that the first course should teach some problem solving techniques and should give the student some laboratory experience. In addition they realize the importance of introducing the student to all the main areas of study. In depth mastery of a subfield of physics is postponed for later courses in which the instructor can assume that the student has already been exposed to the main concepts.

Perhaps we computer science educators should consider the typical first course in physics, as we develop the first course in computer science.

One striking aspect of the typical introductory computer science course sequence is its lack of breadth. Computer science includes among other things a study of hardware, languages, algorithms, data structures, architecture, and operating systems. The recent trend is to incorporate more topics from data structures into the introductory course. But most introductory courses in computer science still neglect hardware, languages, architecture and operating systems.

The physics curriculum generally recognizes that classical Newtonian mechanics is the foundation on which all the other topics can be presented. Consequently, introductory physics courses usually present mechanics first followed by the other topics, which together make up the entire field of physics. Introductory physics courses rarely present just mechanics.

Similarly, the introductory computer science course should begin with algorithm design in a high order language, but not be confined to it. Students should get a better overall picture of the discipline of computer science. Indeed, if they do not get a unified picture of the discipline in their introductory course, where will they get it later?

The thesis of this paper is that a unified view of the discipline of computer science should be presented in the

introductory course, and that the unified view should based on the concept of levels of abstraction. In particular, our beginning students should learn computer systems based on the level structure of Figure 1. The main idea is that computers operate at several levels of abstraction. Programming in Pascal at a high level of abstraction is only part of the story.

To provide a balanced view of the discipline the introductory course should avoid two pitfalls.

First, it should not spend an inordinate amount of time at one level to the exclusion of the other levels. Historically, our tendency has been to spend most (if not all) of our time in the introductory course at level 6, the high order languages level. We then pack the students off to more "advanced" courses that specialize in some other level. It is easy for students to perceive that computer science is a fragmented discipline.

Second, to present a truly unified view of computer science it should not only give more equal weight to each level, but it should emphasize the relationships between the levels. The idea of emphasizing the relationships between the levels of abstraction of a computer system has many ramifications about the introductory course content. It means that the beginning student should understand not only the language at each level, but also how transformations are made from one level to another. The next section discusses some of the ramifications of this idea.

We can achieve the goal of presenting a true introduction to all of computer science in the introductory course only by structuring a tightly coupled introductory course sequence. Again, the model for such a sequence is the typical introductory physics course. The introductory course for physics majors is usually a sequence of two, three, or even four semesters. In the quarter system it usually covers from three to six quarters. The course sequence is tightly coupled in that each course in the sequence is a strict prerequisite for the one that follows, and in that students study from a single text throughout the entire sequence. The sequence is a balanced presentation of all the main areas of physics.

The introductory computer science course should be modeled after that sequence. It should be a tightly coupled sequence of freshman level courses. Each course in the sequence should be a strict prerequisite for the one that follows. Students should study from a single text that places more equal weight on each of the seven levels of Figure 1, and that emphasizes the relationships between the levels.

Compared to the older sciences, computer science is young. Mathematics and physics have been with us for centuries. Computer science has been with us for decades. It is instructive to look at the history of the curricula of these more established sciences.

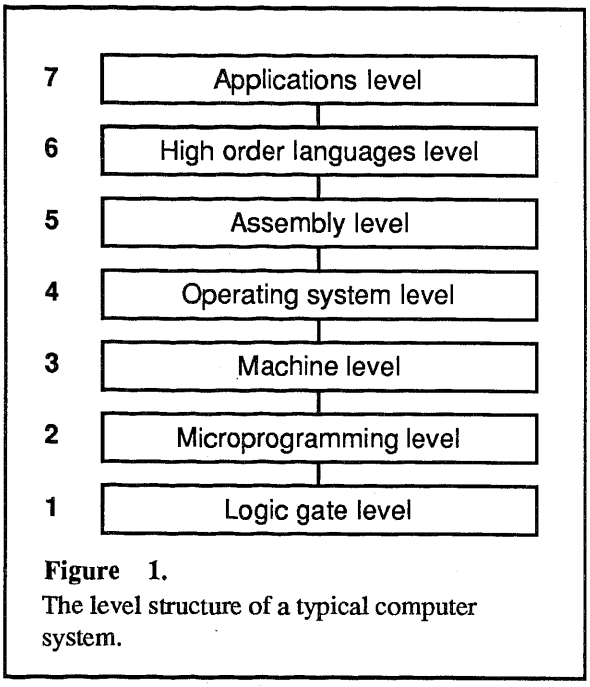

Figure 1.
The level structure of a typical computer system.

In mathematics, the entering university student used to take separate courses in analytic geometry and calculus. Over the years mathematics educators realized that a single course that combined both topics of the subject would serve the student better at the introductory level than separate courses. The change occurred with resistance and debate, but eventually the integrated approach prevailed.

Another lesson of history comes from physics. Separate introductory books and courses used to be devoted to the topics of mechanics, electricity and magnetism, and thermodynamics. Again, after resistance and debate, these topics were integrated into a single course represented by the classic Sears and Zemansky text.

Nowadays there are dozens of introductory calculus and physics texts that are designed to be used in two, three, or even four course sequences. These texts typically run in excess of a thousand pages. Some are printed in two volumes. But the main idea is that they serve a tightly coupled introductory sequence whose goal is to present a unified introduction to the discipline.

The trend of our recent past is towards more integration of computer science topics in the introductory course. In the early days the introductory course emphasized a particular programming language. The language syntax was considered important. The debate often centered on which programming language was best for a student to learn first.

Then came the recognition that software design principles were equally as important, if not more so, that the specifics of a particular language. Books began to incorporate software engineering principles such as

stepwise refinement and top-down design. The concept of structured programming is now well integrated in most introductory computer science books.

The next trend was the incorporation of some data structures into the introductory course. The recent revision of CS1 and CS2 of Curriculum 78 [2, 3] accomplishes this integration. It is appropriate to expose beginning students to data structures concepts because of the intimate relationship between an algorithm and the data structure on which it operates.

This proposal is the logical next step in the integration process. It integrates hardware and software into a complete overview of computer science based on the concept of levels of abstraction. As the integration of data structures was a pulling down of some concepts from more advanced courses, so is the integration of hardware and operating systems concepts. In the same way that including data structures in the introductory course does not eliminate the more advanced data structures course, including hardware and operating systems concepts will not eliminate the traditional computer organization and operating systems courses. Instead, it will give a better foundation on which those courses can be built.

Implementation

This section describes an implementation of the concept. The implementation has three components:

* an introductory computer science course sequence taught at Pepperdine University
* a machine designed to teach the fundamentals of classical von Neumann architecture
* a textbook in preparation based on the course

Seaver College is the undergraduate liberal arts college of Pepperdine University. It is a small college of 2400 resident students, and offers a B.S. degree in Mathematics/Computer Science to about 100 majors.

We planned a major computer science curriculum reorganization during the 1983-84 academic year, and implemented it in 84-85. Part of the plan was the establishment of an introductory course sequence whose goal was to give students a unified view of computer science, with the concept of levels of abstraction as the primary framework. It is a two semester, eight hour sequence.

We encountered two problems in our implementation.

First, one of our goals was to give more equal weight to each of the levels. We wanted to teach students an assembly language at Level 5 in such a way that they would not loose sight of the overall concept of levels of abstraction in the system. The problem was the choice of a suitable architecture and corresponding language to achieve this goal in such a short amount of class time. Our solution was to develop a machine customized for teaching these concepts.

Second, there was no appropriate text on the market. It is not that the material is unavailable. It is available, but it is scattered in many specialized books designed for specialized courses. Our solution was to write our own material for the course. The material is currently being developed for publication as a textbook.

The remainder of this section describes the two-semester course sequence. Space limitations preclude a detailed listing of topics covered with time alloted to each. This information is available from the author.

The course generally presents the levels top-down from the highest to the lowest. For pedagogical reasons we discuss Level 3, the machine level, before Level 5, the assembly level. It seems more natural in this case to revert temporarily to a bottom-up approach so that the building blocks of the lower level will be in hand for construction of the higher level.

The course begins with a description of the concept of levels of abstraction in a computer system. As an example of an application, they learn how to use their text editors in the context of word processing. The skills then carry over to the next part where they must create text files of programs and data.

The first course in the two-course sequence generally follows the CS1 course of Curriculum 78. Our our presentation of problem solving with Pascal is fairly complete. Some topics, such as passing procedures as parameters, are not included.

The course integrates two design methodologies into the Pascal part—stepwise refinement and top down design. Although these two techniques are closely related (some would say identical) we treat them separately. Stepwise refinement is a tool for developing a single main program or a single module. Top down design is a tool for partitioning a program into modules. In this course, the definition of a module is a main program, or a procedure, or a function. One advantage of the levels of abstraction approach is its application in many different areas of computer science. Both of these software design methodologies are based on information hiding and abstraction principles. That principle can be taught here in the context of software design.

One goal of the course is to give the student useful software tools. So we try to present the "best" known algorithms. The sequential file update problem is solved with the balanced-line algorithm [4, 5]. The section on sorting uses the taxonomy of sort algorithms described recently by Merritt [6]. The version of Quick Sort is one that executes in time n log n even in the case when the array is approximately in order originally.

The second course presents PEP/5, a hypothetical computer designed to illustrate computer concepts. Students learn the fundamentals of information representation and computer organization at the bit level. A central theme of this course is the relationship of the levels to one another. Here we show the relationship between the ASCII representation (Level 3) and Pascal variables of type char (Level 6). We also show the relationship between two's complement representation (Level 3) and Pascal variables of type integer (Level 6).

The PEP/5 computer is a classical von Neumann machine. The CPU contains an accumulator, an index register, a base register, a program counter, a stack pointer, and an instruction register. It has four addressing modes—immediate, direct, indexed, and stack relative. At this point, students can run short programs in binary on a simulator. The PEP/5 operating system is in simulated read only memory (ROM). It can load and execute programs in hexadecimal format from students' text files. They learn by running the simulator that executing a store instruction to ROM does not change the memory value.

The native machine instruction set contains one input and one output instruction. These are single character instructions based on the ASCII character set. A group of four instructions have unimplemented opcodes that generate software interrupts. At the assembly level the operating system provides the four instructions as decimal and hexadecimal I/O.

Level 5 is the assembly level. The course presents the concept of the assembler as a translator between two levels. It shows the relationship between assignment statements at Level 6, and load and store instructions at Level 5. It introduces Level 5 symbols and the symbol table.

The unified approach really pays off here. Students learn a specific Level 6 language, Pascal. They learn a specific von Neumann machine, PEP/5. This part continues the theme of relationships between the levels by showing the correspondence between:

* loops and if statements at Level 6, and branching instructions at Level 5
* arrays at Level 6 and indexed addressing at Level 5
* procedure calls at Level 6 and the run-time stack at Level 5
* function and procedure parameters at Level 6, and stack relative addressing at Level 5
* case statements at Level 6 and jump tables at Level 5.

The beauty of the unified approach is that the course can implement many of the examples from the Pascal part at this lower level. For example, the run-time stack illustrated in the recursive examples of the Pascal part corresponds directly to the hardware stack in PEP/5 main memory. Students learn the compilation process by being able to translate manually between the two levels.

This approach also provides a natural setting for the discussion of central issues in computer science. For example, the course presents structured versus unstructured flow of control in the context of programming at Level 6 versus programming at Level 5. It discusses the goto controversy and the structured programming/efficiency trade-off with concrete examples from languages at the two levels.

Curriculum guidelines warn against a common pitfall of the introductory assembly language course. The danger is that the student will get mired down in the details of a specific machine and will not be able to cull the fundamental principles from the material. Most assembly language books simply do not assume a common high order language background of the student. Therefore they are never able to make as strong a connection between the levels of the system as this course does.

The unit on language translation principles is motivated by the goal to emphasize the relationships between the levels of Figure 1. It introduces students to simple parsing theory. It begins with a formal definition of phrase structured grammars, and describes context-sensitive, context-free, and regular grammars. It shows the relationship between regular grammars and finite state machines. Furthering the goal of providing useful software tools, the course shows how to implement finite state machines in Pascal as recognizers. It presents a Pascal program that contains a finite state machine and translates between two small languages.

Here is yet another advantage of the unified approach. Students learn a specific high order language, Pascal. They learn a specific assembly language, PEP/5. They learn a specific tool, the finite state machine. The translation program as a model now equips them to write a small PEP/5 assembler in Pascal. That project is a key element of the course. It illustrates in a concrete way the relationship between two different levels of abstraction.

The last topic is a description of the function of an operating system and a few of the key issues in operating system design. Two topics, one on loaders and another on interrupt handlers, are illustrated with the PEP/5 operating system. The system has a simulated ROM burn-in facility. Students can rewrite any part of the operating system, assemble it, "burn it" into ROM, install it, and run it on the PEP/5 system. They can redefine any of the unimplemented opcode instructions.

Although a complete survey would take the student down to Levels 2 and 1, we have found that to not be possible in eight semester hours. However, we believe that our course is a step in the right direction toward giving our students a unified view of the discipline of computer science.

Our course structure does not follow all the CS 1 and CS 2 guidelines of Curriculum 78. Although we cover all

of CS 1, we omit the following topics from CS 2: program specification, queues, random access files, array implementation of linked lists, array and record implementations of trees, linked list implementation of sets, hashing. These topics are postponed until the data structures course. We introduce the student to the concept of a data structure, with the example of the stack. That is the data structure necessary to understand the implementation of recursion and process interupts at the lower levels.

Conclusion

Gibbs and Tucker [1] define computer science as the systematic study of algorithms and data structures, specifically

(1) their formal properties,
(2) their mechanical and linguistic realizations, and
(3) their applications.

They distinguish computer engineering as that discipline in which (2) takes precedence over (1), and information systems in which (3) takes precedence. Their view of computer science is that discipline in which (1) takes precedence. This paper argues for an introductory view of computer science in which (1) and (2) take equal precedence.

Although the effort by Gibbs and Tucker is to define a computer science curriculum appropriate for a liberal arts degree in computer science, I believe that an introductory course that follows the philosophy outlined here is applicable as a foundation course for both liberal arts colleges and engineering schools. The situation is similar to the other pure and applied sciences. Mechanical and electrical engineering courses assume that their students have taken an introductory physics course. Chemical engineering students start with introductory chemistry. The introductory course services both chemistry and chemical engineering majors. Likewise, the introductory computer science course should service both computer science and computer engineering majors.

Computer science is remarkably young. Its curriculum is in a continual state of flux compared to the more established sciences. As the discipline matures, so will the curriculum. We have every reason to believe that the curriculum will evolve to a state resembling that of the more established disciplines.

Acknowledgements

The book, *Structured Computer Organization*, by Andrew S. Tanenbaum [6] has influenced my thinking about the curriculum more than any other. This paper extends the level structure of Tanenbaum's book by adding the high order programming level and the applications level at the top. Many people contributed to the ideas of this paper. Don Thompson, Don Hancock, Carol Adjemian, and Chelle Boehning taught sections of the course described here, and contributed many suggestions for improvement. Pepperdine University in the person of Ken Perrin provided the creative environment in which the idea behind this project was able to evolve.

References

[1] Gibbs, N. E. and Tucker, A. B., A Model curriculum for a Liberal Arts Degree in Computer Science, *Commun. ACM 29*, 3 (March 1986).

[2] Koffman, E. B., Miller, P. L., and Wardle, C. E., Recommended Curriculum for CS1, 1984: A Report of the ACM curriculum committee Task Force for CS1, *Commun. ACM 27*, 10 (Oct. 1984).

[3] Koffman, E. B., Stemple, D., and Wardle, C. E., Recommended Curriculum for CS2, 1984: A Report of the ACM curriculum committee Task Force for CS2, *Commun. ACM 28*, 8 (Aug. 1985).

[4] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.

[5] Levy, M. R., Modularity and the Sequential File Update Problem, *Commun. ACM*, (June, 1982).

[6] Merritt, S. M., An Inverted Taxonomy of Sorting Algorithms, *Commun. ACM*, (Jan. 1985).