9. Theory and Practice in the Fundamental Computer Science Course

The fundamental computer science course in this curriculum is the two-semester sequence FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212]. Fundamental Structures was originally designed in 1974 and 1975, and the original lecture notes evolved to a textbook based upon the course [127]. In 1978 William A. Wulf, Mary Shaw, Paul N. Hilfinger, and Larry Floll prepared two papers about the course [36, 50]. This chapter incorporates extensive material from these two papers along with new information based on more recent offerings of the course.

Traditionally, the first two programming courses have emphasized basic techniques and skills—the details of a programming language, basic problem solving and program development, “structured programming,” the manipulation of simple data structures and files, basic sorting and searching algorithms, etc. They have placed little or no emphasis on such “advanced” or “theoretical” material as rigorous specification and verification, formal language definition, automata, or performance analysis. Their approach, in other words, is similar to that taken by elementary calculus courses, which teach the mechanics of differentiation and integration without bringing in too much theoretical material. The reason for avoiding theory in an elementary calculus course is quite legitimate: for most students, rigorous treatment is not useful enough to justify spending time on it at the expense of manipulative skills. We shall argue here that this reasoning does not apply to programming.

If computer programming is to become an “engineering discipline,” computer science students must acquire the tools for rigorous analysis and evaluation of programs. An early course in the undergraduate curriculum should introduce the fundamental principles of the discipline, just as freshman calculus teaches the basic analysis skills for most engineering disciplines. The course described in this chapter teaches mathematical principles and practical programming applications in a unified form.

9.1. Introduction

The typical undergraduate computer science program begins with introductory courses in programming, computer organization, and data structures. Subsequent courses in comparative languages, compilers, programming methodology, and systems programming contribute to a student’s programming ability. Although students are generally required to learn some theory (e.g., discrete mathematics and automata theory) they rarely appreciate, as undergraduates, the relationships between the theoretical foundations of computer science and the programming process.
Yet the study of the programming process has come to be known as "software engineering." The term "engineering" is not used lightly, but to convey the idea that good software can be constructed only through the application of sound scientific principles that lead to simple, understandable structures, cost effectiveness, and reliability. Since engineering is the application of science, software engineering should be the application of computer science. How, then, can software engineering be properly taught if students do not understand the underlying principles?

FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212], taken by students after they have completed an introductory programming and a discrete mathematics course, is the course in this curriculum that expose undergraduates to theory at the same time as relating that theory to programming. It is a prerequisite for most other computer science courses. It integrally includes not only topics from conventional second and third computer science courses (data structures, "structured programming", programming language details, etc.), but also basic theoretical material. These topics include automata and formal languages, formal specification of algorithms and data types, complexity analysis, program and data abstraction techniques, and program verification. Naturally, we introduce these topics at an elementary level. Nonetheless, they form the central organizing core of the course. In a sense, this course is as much about reading programs as it is about writing them. The course has been offered in various forms for ten years, and our experience shows that most students can handle the more advanced material.

Some important benefits have resulted. First, students become familiar with important mathematical tools, including logic and sets, induction and recursion, and the concepts of formal proof and mathematical model. Although the familiarity is one of manipulative ability rather than deep theoretical understanding, it is nonetheless quite valuable as a foundation.

Second, this course teaches important prerequisite material for more advanced courses. For example, a course on compilers needs the concepts of grammar and finite state automaton, rather than a deep theoretical understanding of them. Since the concepts of state and state transition are important to other aspects of programming, we now introduce them much earlier. The instructor of a compiler course may find a brief review in order, but the emphasis can be placed on lexical analysis and parsing, not on automata themselves.

Third, because practical and theoretical topics are presented together, it is easier to show how the same ideas reappear in different forms in different areas of computer science.

Overall, our principle motivation is the conviction that programming should be an engineering discipline, and that engineering disciplines must be grounded in science. Further, to be effectively taught, this science must be