Proceedings of the Ninth Technical Symposium on Computer Science Education, ACM SIGCSE Bulletin, 10(1), 255-259, 1978.

August 1978

# INTRODUCING "THEORY" IN THE SECOND PROGRAMMING COURSE

Paul N. Hilfinger, Mary Shaw, Wm. A. Wulf
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Lawrence Flon
University of Southern California
Los Angeles, California 90007

## 1. Introduction

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 complexity 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 foundation material and technically rigorous treatments. 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. In this paper, we shall argue that this reasoning does *not* apply to programming.

Since the mid-1960's, and the advent of the "software crisis", there has been a definite trend away from teaching just the basic skills of programming. Text books and courses have placed increasing emphasis on the *engineering* considerations involved in good programming, on style and program structure. The new ACM curriculum recommendations [1] reflect this trend. We have come to realize that these engineering considerations are as important to programmers as are the details of particular programming languages or particular algorithms.

At Carnegie-Mellon University, we have been experimenting for four years with a further extension of this trend. We have introduced basic material from certain "theoretical" topics as integral parts of our second programming course. These topics include automata, formal specification of algorithms and data types, complexity analysis, and program verification. Naturally, we introduce these topics at an elementary level. Nonetheless, they form the central, organizing core of the course.

Our principle motivation is the conviction that programming should be an engineering discipline, and that engineering disciplines must be grounded in engineering *science*. Further, to be effectively taught, this science must be introduced as early as possible. In contrast to the measure-theoretic underpinnings of the calculus, the theory we introduce is *immediately useful* to working programmers. In addition, the topics we cover are fundamental to later courses. By introducing the topics early in the curriculum, we provide a common vocabulary for these later courses, eliminate redundant treatment of topics, and give students greater exposure to the material and a better chance to absorb it.

We feel we have been successful with the course, although our evidence is necessarily subjective. Because its philosophy flies in the face of current practice, we shall attempt to explain that philosophy in this paper. The course itself, "Fundamental Structures of Computer Science" (FS hereafter) is described in somewhat greater detail in [2] and we are preparing a supporting text [3]. We are not so much interested in pressing our own particular treatment of the topics, however, as we are the *choice* of material and its place in the curriculum.

## 2. The Setting

Most of the students taking computer science courses at Carnegie-Mellon come from engineering (especially electrical engineering) and applied mathematics. Their first programming course covers most of the material outlined in CS 1 and CS 2 of [1], but places fairly light emphasis on the later sections of CS 2 -- internal sorting and searching, data structures, and recursion. Students may take either a FORTRAN or a PASCAL version of this first course. We recommend the Pascal version for those students interested in the later computer science courses, especially FS, as Pascal is currently the language used in those courses. However, for various reasons, a substantial fraction of the students in FS come from the FORTRAN course.

In the past, many of our students have had little exposure to discrete mathematics. The FS course requires nothing more than familiarity with standard set theoretic and logical notation and definitions, rather than a complete course. Still, we consider it undesirable to spend time teaching these topics (as we must at present), since various portions of the material are already familiar to a fairly large percentage of the class.

The FS course takes the place of the usual data structures course (CS 7 of [1]). Subsequent courses' topics include software engineering (aimed at giving students experience in writing large software and familiarity with the problems), comparative programming languages, computer architecture and hardware, real-time and minicomputer programming, graphics, artificial intelligence, and compiler and operating system design. Thus, FS must provide the necessary preparation for these courses.

## 3. Theoretical Content of the FS Course

In the following sections, we enumerate the particular topics we have covered, giving our reasons for including each. We are not attempting to outline the actual FS course here (see [2]), although we shall refer to it often. For one thing, we have varied its content from year to year and have not always included each of the topics we shall discuss.

### 3.1. Models of Computation: Automata

The FS course discusses the notion of an abstract model of computation and introduces several of the standard models: the finite state machine, the push-down automaton, and (very briefly) the Turing machine. The discussion provides a framework in which to discuss models in general and their use in understanding abstract properties of programs. For example, we discuss the relative power of programming constructs, and show how one can perform such comparisons having suppressed inessential details. Finite state machines also provide a setting in which we can introduce non-deterministic computation early and gently.

In practical programming terms, the finite state machine is the skeletal prototype of the table-driven computation. The FS course, for example, takes students through the construction of a table-driven lexical analyzer. Traditionally, lexical analysis is the subject of a compiler course, but an applications programmer who wishes to write sophisticated input routines may well find the technique useful. Finite state machines also appear in certain recent algorithms for string matching.

Knowledge of automata is of obvious use in later courses on compilers and in understanding sequential logic design. In particular, we find it convenient for students to have seen the concept of the *state of a computation* and of state transition before going on to a compiler course.

### 3.2. Formal Languages

At an elementary level, the study of formal languages provides two descriptive notations -- regular expressions and BNF. Not only are these useful in defining the syntax of programming languages (and hence, in reading programming language definitions), but they are convenient and succinct notations for describing many other notations — program input formats, for example.

As specification tools, both BNF and regular expressions have the advantage that there are well-established techniques for obtaining implementations (parsers) from the specifications. Even without going into the theory of parsing, the FS course still lays much of the groundwork for these constructions by surveying the correspondences between automata and grammars. We first specify the lexical analyzer mentioned in the last section, for example, using regular expressions. This specification guides the construction of the tables.

BNF provides many examples of recursive definition. Thus, it serves as one introduction to the topic of recursion (and certainly a better motivated introduction than the recursive factorial function). The structure of a simple recursive descent parser follows naturally from the grammar being parsed, providing a neat transition from recursive definition to recursive implementation.

### 3.3. Formal Specification and Verification

One of the most important lessons to be learned in any introductory programming sequence is the distinction between specification and implementation — between what is to be done and how it is to be accomplished. Our own experience as programmers indicates just how important this distinction is: we cannot *conceive* of large programs until we shake off the details of their construction. At the same time, our teaching experiences also show that the idea of rigorously specifying a program apart from its implementation and the concomitant practice of using the specifications of a routine rather than its code to understand its effect, are among the most difficult things to teach undergraduates. Hence, we consider it vital to stress rigorous specification and the distinction between specification and implementation as early and often as possible.

Therefore, the FS course introduces the use of mathematical entry and exit assertions for programs and procedures. Besides their utility as a specification device, these have the pedagogical advantage that the student is forced to abstract and to write a general statement of the effects of his program, since it is extremely awkward to encode the workings of a program in its entry and exit assertions. As the FS course now stands, most of the practice that students get in using formal specification comes in the units on data types (see section 3.5).

We cannot expect students to become proficient in the art of program specification as a result of the FS course alone. It is a difficult art, requiring experience and, ideally, a certain mathematical tastefulness for its successful practice. This is true even when the specification language is precise English (as it often must be) rather than mathematical notation; the difficulties of producing good documentation are well known. Nonetheless, we find in later courses that our students have started thinking in terms of the abstract effects of their programs.

Having introduced program specification, one can start thinking about verifying the correctness of an implementation with respect to a specification. Again, the aim cannot be proficiency, but rather inclination. We want to show that it is *possible* to argue rigorously and systematically about various properties of a program. Our intent is to encourage students to understand their programs as rigorously as possible. Whether or not one believes that detailed formal program verification is feasible in practice, it is still valuable to acquire the habit of verifying programs *informally* and of constructing them to make that task easier. We can show, for example, that not only does the proper modularization of a program into routines make them more readable, but it also simplifies formal and informal reasoning (verification conditions become smaller).

Standard treatments of verification tend to stress mechanical manipulation - the generation of verification conditions. It is true, furthermore, that such concrete skills are easier to teach (if harder to motivate) than general methodological principles. Still, teaching the general principles, such as "rigorous program construction", is really our global aim. The purpose of introducing program verification at all is simply that a program is generally cleaner, simpler, and more likely to be correct if it is well thought out and carefully constructed. Even if a student never again formally verifies a program, simply understanding that it *can* be done will change the ways he thinks about programs and programming.

## 3.4. Algorithmic analysis

Cost analysis is a critical activity in any engineering discipline. Programming is no exception. From personal experience, we know that careful *quantitative* analysis of program cost is all too rare. Many programmers "optimize" programs only at the "put as much as possible into registers" level, rather than at the algorithm level, and have no good idea of where their programs can actually benefit from optimization.

Therefore, the FS course introduces some simple cost analysis techniques. It discusses the notion of the *order of complexity* of a computation, and the distinction between the shape of a cost curve and its actual values. All of this material is rather obvious, of course, but until it is pointed out that cost analysis might be interesting, programmers tend not to do it. By introducing students to the subject formally, we get them in the habit of making implementation decisions on the basis of expected costs, and to develop in them an ability to do "seat of the pants" performance estimates. We need hardly mention that these techniques and attitudes are useful in any later course on the analysis of algorithms (e.g., CS 14 of [1]).

## 3.5. Data Types

As we mentioned above, the FS course subsumes the usual data structures course. However, its treatment of data structures differs from what we observe to be the standard. Traditionally, individual data structures are introduced in close association with their standard representations. In keeping with current methodology, as well as our policy of *separating* specification from implementation, the FS course first introduces the general concept of a *data type*. That is, we define a data type in terms of the legal operations and the essential observable behavior of those operations. We go on to show how to specify this observable behavior rigorously, and to use the data types thus described, without mentioning how the operations are implemented.

Having developed the tools needed to describe data types, we introduce the standard data structures in terms of their abstract properties. Later, and separately, we discuss standard implementation techniques. This allows us to explain several alternatives and discuss their relative merits. Moreover, it emphasizes that many variations are legitimate; there is no distinguished representation of a linear list, for example.

The *separation of concerns* behind this approach has proved valuable in developing large software. As was the case with the specification of procedures, however, novice programmers initially have difficulty understanding that a data type has an abstract behavior distinct from its implementation. This only suggests to us that they should start getting used to the idea early in their programming education.

We have found a fairly simple exercise that illustrates the distinction between specification and implementation. We first asked the students to provide the implementation of a simple data type (e.g., <u>stack</u>) using a particular representation (e.g., the standard vector representation). Next, we asked them to write a main routine implementing a simple algorithm which uses data of this type (e.g., an iterative routine to convert Polish notation to infix form). They were to code this algorithm using only the abstract specifications of the data type. Finally, we asked them to change to a different implementation of the data type (e.g., using a linked list representation) without modifying a single character of the main routine.

This exercise, simple as it seems (and <u>stack</u> is the archetypical "toy" data type), proved quite useful in illustrating the point of data type abstraction. It is particularly convenient if the course uses a language like PASCAL. However, it is possible in FORTRAN as well.

## 3.6. Recursion

At least one recent paper [5] argues that recursion is not simply an esoteric device, but a programming tool which can facilitate algorithm development. Instances crop up quite naturally, such as recursive descent parsing from a BNF definition, which we mentioned earlier. One important class of instances is the set of divide-and-conquer algorithms, whose natural descriptions are recursive. Several sorting algorithms, for example, have a high-level divide-and-conquer description. Finally, of course, recursively defined data structures (such as trees) are most naturally manipulated recursively.

In the FS course, we try to present what might be called "recursive thinking" -- the art of seeing appropriate problems in terms of smaller instances of itself. It is its own formal verification method. One simply assumes (inductively) that all recursive calls behave according to specification and shows that the arguments of recursively nested calls get progressively "smaller". This kind of reasoning occurs throughout a course on algorithm design or compiler design. Its early presentation therefore serves as important preparation for these later courses.

Recursion is another place where the specification and implementation of a program may differ. We may specify a program recursively, but implement it as an iterative program (perhaps with a stack). For example, merge sorting has a very simple recursive definition, although its usual implementations are iterative and interleave the recursive calls. This brings up the whole subject of program development by successive transformation. The similarity between simple push-down automata and simple recursive programs, in particular the fact that one may use either to recognize context-free languages, provides one starting point for such a discussion. This topic is one which we will cover in a future version of the FS course.

## 4. Experiences

Over the last four years, the bulk of the material above has appeared, with varying relative weights on the topics, as part of the FS course. The course also involves most of the usual data structures material and requires a considerable amount of programming practice of the students. Roughly four hundred students have taken the course under a total of seven different instructors. Again, we point out that it is a second course, which most students take at the beginning of their sophomore year.

Perhaps the natural reaction, or at least the one we hear from some colleagues, is that the course is too rigorous, too "formal", and too voluminous for its sophomore audience. Students, however, are generally favorable, and show up in later courses ready to use the material. One purpose in writing this paper is simply to announce that our experience shows that students *can* handle the material. Indeed, they perceive the bulk of the material to be quite elementary. We simply take care never to suggest that, e.g., program verification and recursion are *supposed* to be difficult topics.

Our students have had some difficulty with the *volume* of material. We expect to relieve some of this difficulty by eliminating our initial lectures on discrete mathematics, and reorganizing the course description and requirements to reflect the change. In any case, we need very little discrete mathematics; students who find themselves very weak in that area could easily pick up the necessary notations and definitions from such texts as [4]. It also appears that the FS course functions well as the first in a two-course sequence. The second course in the sequence is a course in software engineering in which students can practice applying the principles presented in the FS course to reasonably large problems. This second course can relieve some of the obligation of the FS course to provide the experience, and evens out the students' work loads.

## 5. Conclusion

The topics we discuss in the FS course contain nothing that we (the authors) don't use ourselves as programmers. The FS course provides a common basis for later courses, which in itself is useful. Yet we intend that, ideally, *all* the material we cover should be part of any programmer's "bag of tricks".

We do not view the FS course as a particularly radical departure from the *subject matter* of [1] or similar curriculum proposals. We do think it an important re-ordering of the material. It is an attempt to present basic principles together in one course and to teach these principles *before* their use in more advanced courses. Thus, just as in engineering one teaches mechanics before structural design, we have put the science and mathematics of programming *before* most of the actual coding.

We have tried to present a set of theoretical topics which we believe to be valuable in the practice of software engineering. We have tried to argue that students should learn these topics as early as possible. In short, we want to promote the notion of programming as an engineering discipline employing scientific and mathematical methods.

## 6. References

[1] "Curriculum recommendations for the undergraduate program in computer science", SIGCSE Bulletin 9,2 (June 1977).

[2] Lawrence Flon, Paul N. Hilfinger, Mary Shaw, and Wm. A. Wulf. "A fundamental computer science course that unifies theory and practice." Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education, Detroit. SIGCSE Bulletin 10, 1 (February 1978), pp. 255-259.

[3] W. A. Wulf, M. Shaw, L. Flon, and P. Hilfinger. Fundamental Structures of Computer Science. Textbook in preparation (Addison-Wesley).

[4] Donald F. Stanat and David F. McAllister. Discrete Mathematics in Computer Science. Prentice-Hall, Inc., Englewood Cliffs, 1977.

[5] Zohar Manna and Richard Waldinger. "Structured programming with recursion." Memo AIM-307 (STAN-CS-77-640), Stanford Artificial Intelligence Laboratory, January, 1978.