METHODOLOGY FOR TEACHING INTRODUCTORY COMPUTER SCIENCE

R.R. Oldehoeft
R.V. Roman

Department of Mathematics
Arizona State University

INTRODUCTION

In the last few years it has been generally recognized that teaching programming involves more than describing a new FORTRAN statement each day and providing programming problems to be coded. The concepts of disciplined programming and the accompanying interest in the problem solving process, coupled with the increasing economic desirability of constructing correct and maintainable software has resulted in significant attention being focused on what should be taught, and, to a lesser extent, on how this is best accomplished.

In order to discuss how an introductory programming course is taught, it is essential to first establish a set of objectives for such a course. Some of these objectives are obvious and generally agreed upon, others are engendered by the academic environment in which this particular course exists. We first state the objectives and then comment on their appropriateness and interrelation.

1. The student should gain a knowledge of the basic problem solving methodologies applicable to developing programmable solutions to problems.

2. The student should gain skill in a locally usable high level programming language and be able to construct understandable programs in this language.

3. As an introductory course, this course should introduce novices to what constitutes computer science and provide basic knowledge for subsequent coursework in the area.

4. As an introductory course, students who have no prior knowledge of computer science and find it not consistent with their abilities should be able to survive the course with moderate effort and without special treatment by the instructor.

The remainder of this paper addresses the question of how an introductory programming course can be taught to realize these goals. The appropriateness of the first two objectives for any introductory programming course is generally agreed upon [Gri74]. Teaching only a programming language with no attention to the basic problem solving methodologies implies that either problem solving is unimportant or intuitive, neither of which is true. Teaching only abstract problem solving and either no programming language or one which is arcane, impractical, or foppish, implies a misconception of the mental capabilities of the average undergraduate or a complete disregard for practicality. Thus the first two objectives are consistent and, in themselves, appear to generate no conflicts; the student should learn general problem solving techniques and apply them to developing programs in a practical manner.

The last two objectives are not as general with respect to introductory programming courses but are necessary due to the particular academic environment in which this course has been developed. The course serves as a first course for computer science majors and consequently is an introduction to the curriculum. For this reason it must present foundation material for subsequent courses and must also present a picture of what constitutes computer science. The design of subsequent courses for computer science majors may assume that students who enroll are interested in the field and have more than an average aptitude for the material. This is not the case in an introductory course and consequently allowances must be made for students who are taking the course due to requirements in another major or those who enroll to learn what computer science is and during the course find it inconsistent with their abilities or interests.

The third goal moderates the single-mindedness with which the first two goals may be pursued. Topics other than problem solving and programming must be presented to give an overview of computer science and the skills which the student gains with respect to the first two objectives must be applicable in future courses.

The last goal implies that whatever is done with respect to achieving the first three goals must take into account the student of marginal aptitude. Large intuitive leaps must not be required for the average student to make reasonable progress. Further, successful completion of the course should leave the student with skills useable without further coursework.

In the following section we discuss a general scheme for teaching elementary problem solving and as associated programming language. In subsequent sections we discuss the major decisions that must be made in implementing this general scheme for a specific problem solving notation and programming language. A final section summarizes the major features of the scheme and its implementation as well as its observed success.

THE GENERAL SCHEME

First Phase: Precise, Sequential Thinking

The first phase of teaching elementary problem solving applicable to program development involves capitalizing on the problem solving ability already present in the student. During this phase the student is presented with an absurdly (possibly insulting) simple problem involving no decisions. For example find the sum of squares of two numbers or find the average of three numbers. The problem is initially presented with specific inputs (find the sum of squares of 4 and 7) for which specific results are required. It is then generalized and the notion of a variable is naturally motivated (find the sum of squares of A and B) along with the concepts of giving variables initial values (input), computing new values from existing values (simple assignment) and specifying results (output). During the presentation of such examples the necessity of precise, sequential thinking is strongly and repeatedly emphasized and a written notation for expressing a linear sequence of precise steps is developed.

It is at this point that elementary statements from the programming language, now well motivated, may be introduced. The student is given a set of simple clerical rules for translating individual steps in a problem solution to statements in the chosen programming language.

The final exercise of this first phase involves assigning a problem which requires computing several simple results from a small set of input values (e.g., given 4 values, compute their sum, the difference of the product of the first two and the quotient of the last two, etc.). The student is required to firt develop a written description of the problem solution, which is strictly graded for preciseness, and is then required to translate the solution and run the resulting program. In order to allow an arbitrary number of data sets to be handled, we introduce the notion of a closed reading loop along with the language features necessary to implement it. While the student is developing this very simple program, debugging techniques are introduced, including verification of input values, and interrogation of intermediate results. A final step in this programming assignments is for the instructor to supply the student with test data once the student has concluded that the program is correct. This is an excellent opportunity for the instructor to convince the student that even the simplest program is seldom thoroughly debugged (provide input so that a division by zero will occur, or provide badly formatted input that will yield incorrect results) and that extreme precision is required to assure that the resulting program does solve the class of problems it is purported to.

Second Phase: Disciplined Control

The second phase of teaching elementary problem solving involves introducing the student to a small but sufficient set of control constructs. As each construct and its notation are introduced, the programming language features which implement it are defined and a precise clerical scheme for translating from the notational form of the construct to the programming language is presented as dogma, i.e., deviation from the standard translation is a serious error in graded work. The order in which the selected control constructs are motivated and presented is determined by how natural they are to the student's existing problem solving patterns. It is felt by the authors and reflected in the ease with which the student's grasp the various constructs that selection constructs (IF-THEN-ELSE, and CASE) are quite natural, while iteration constructs (WHILE-DO, REPEAT-UNTIL) are not intuitive. This observation and the pedagogical desirability of keeping the set of allowable constructs as small as possible suggests that IF-THEN, IF-THEN-ELSE and WHILE-DO are appropriate initial choices. CASE is introduced much later as a convenient alternative to an unwieldy collection of nested IF-THEN-ELSE constructs. REPEAT-UNTIL is introduced by posing a problem where an iterative construct with trailing decision is both natural and proper (e.g., the Euclidean algorithm).

IF-THEN and IF-THEN-ELSE are easily motivated with problems only slightly more complex than those used in the first phase. For example, find the largest of three values, compute the average of two values if the first is greater than the second, otherwise compute their sum. The student's understanding of these constructs is assured through numerous problems for which they are required to develop structured solutions using only those constructs available. Care is taken to assign problems whose structured solutions are natural and straight-forward, leaving problems where purely structured solutions are awkward for discussion only. For example, problems which yield to the CASE construct when only IF-THEN

and IF-THEN-ELSE are allowed, and problems whose natural solution involve three cases where there is substantial overlap between, say, cases 1 and 2 and also between cases 2 and 3 should be avoided.

Most students grasp the concept of these selection constructs quite readily. It should be noted, however, that a substantial number of students have difficulty with the mutally exclusive property of the IF-THEN-ELSE construct, i.e., problem solution steps in the ELSE portion of the construct may be written assuming the condition tested is false. For example, the following is typical:

```
IF    k = 0 THEN

        ...

ELSE

  IF k ≠ 0 AND a < b THEN

        ...
```

For most students this is corrected with careful grading of problem solutions.

Unlike selection constructs, iterative constructs are not easily recognizable in the students mental problem solving process. When asked to analyze and make precise the mental algorithm for finding the largest value in a list, few students recognize the iterative nature of the search; they feel it is sequential. This is not particularly surprising in that any search that the student performs addresses a list whose length is known and the search is sequential. Similar observations can be made with respect to the mental algorithms we use for computing the sum of a list of numbers (we use no accumulation variable) or sorting a list (we do not think in terms of interchanges). Consequently the notion of iteration must be taught rather than just made precise as in the case of IF-THEN and IF-THEN-ELSE.

The approach used is to, once again, select an absurdly simple problem which admits an iterative solution. Initially a sequential solution is developed and then analyzed to detect a situation where iteration will greatly simplify or, more importantly, generalize the solution. For example, consider the problem of printing the first 5 integers. The sequential solution is

```
PRINT 1

PRINT 2

PRINT 3

PRINT 4

PRINT 5
```

The student is then asked to observe that there are five statements that vary only slightly. The nature of the variation is then carefully examined. This motivates the concept of a control variable as well as its initialization, incrementation and testing and yields the iterative solution

```
I ← 1
WHILE I ≤ 5 DO
  PRINT I
  I ← I + 1
```

Next the problem of printing the first 100 integers is addressed and finally the general problem of printing the first N integers is posed and solved quite simply with iteration; no sequential solution exists.

This process is repeated several times on problems of increasing difficulty and the student is required to develop iterative solutions for a collection of problems of varying difficulty.

Given that iteration is accepted as a difficult concept requiring teaching it is pedagogically unwise to clutter its presentation with other new concepts. There are many problems of vastly varying difficulty which involve only scalars in their definition and solution. It is neither necessary nor desirable to simultaneously introduce arrays and iteration. Problems using only scalars do allow for the introduction of the concepts of an accumulation variable and a flag variable through motivating examples and exercises.

Third Phase: Data Structures and the Top Down Approach

Once the student has been thoroughly familiarized with the basic collection of control constructs, it is then appropriate to introduce elementary data structures. The linear array is introduced by first motivating its need through problems not solvable with scalars. The concept of the array and the accompanying notation are presented in the environment of a model where precise, pictorial description of array declaration and indexing are possible. After several example problems are solved, the student is assigned a problem set which includes array problems of varying difficulty, to be solved but not programmed. This problem set includes finding the largest value in an array, removing duplicate values from a sorted array, merging the values in two sorted arrays into a single, third array (the student is warned that a structured solution to this problem is more difficult). This problem set is graded for correctness (for incorrect solutions the instructor provides input data for which the solution fails) and proper structuring, both weighted equally.

At this point the student is prepared to appreciate the notion of top down development of problem solutions. The concept of a precise problem solution step which has specific inputs and requisite outputs, but requires further work to develop its final, programm-

able for is familiar. The teaching of the top-down approach and step-wise refinement has been discussed elsewhere [Con73]. Our approach is comparable.

PROGRAMMING LANGUAGE CONSIDERATIONS

In this section issues concerning teaching introductory computer science are discussed in relation to the programming language used. Our approaches do not dictate the choice of a particular language and general discussions regarding this question have been adequately presented [Smi76]. The reasons for our choice of FORTRAN IV are included in the following.

Language and Course Objectives

The objectives of this course influence the choice of programming language and, more importantly, how the language is presented.

1. The first objective requires that the language be algorithmic in nature and should have control structures like those used by students in the problem solving phase that precedes coding. The well known deficiencies of FORTAN are taken up in the next section.

2. Availability and high level are the only constraints derived from the second objective; any of a variety of languages will do.

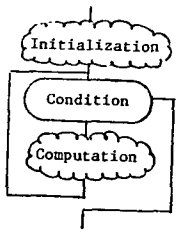
3. Since the course is followed by further course work, the programming language should provide a tool that is usable there. FORTRAN has been maligned in this regard because of its lack of interesting inherent data structures. Two points should be noted. First, FORTRAN is the initial language not only one a student will learn. After having seen several languages, students who are given projects to complete using a language of their choice often make an appropriate decision. FORTRAN is neither clung to nor abandoned. Multilingual approaches have also been common. Second, there is a difference between learning the nature of data structures and learning data structure applications. Implementation of e.g., lists via encapsulating procedures written in a language without list structures teaches of the disadvantages of the structure as well as the power.

4. Separation of service courses and introductory computer science brings clear advantages. Disadvantages exist as well. Staffing is a major problem. A student in a service course who elects to pursue further computer science study is inadequately prepared. Since this course must serve those who are beginning a computer science curriculum, those who are undecided, and those whose main interest lies elsewhere, the programming language should be easy to master should be amenable to incremental teaching, and should provide a useful tool for those who do not study additional computer science. Since this course serves mainly science and engineering students, FORTRAN is the overwhelming choice of most departments. Computer science students will learn several more languages; keeping them ignorant of FORTRAN would be a serious disservice.

To summarize, a programming language should be simple, subsettable, high level, algorithmic and available, should possess appropriate control structures, should be one of the tools an advances student will need and should be a suitable single language for people outside of computer science. No such language exists. The method of presentation of an actual programming language can ameliorate deficiencies and achieve objectives.

Language and Problem Solving Notation

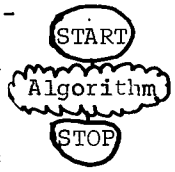
The notation used for problem solving in this course is the flowchart. Other notations possess the advantage that construction of badly structured algorithms is impossible. Our choice was based on the ubiquity and universal understanding of flowchart notation. The flowchart forms used in problem solving are restrictred to a few well-known structures. A major aim of the course is convincing the student that the algorithmic solution expressed using these forms is the most important product. However, the implementation and computer execution of the algorithm is an effective test, and student interest lies in the production of correct programs. To that end, the translation to a programming language should be simple, consistent, and not liable to introduce additional errors. Mechanical translations to FORTRAN are provided for each form; this translation is always required. For example, the WHILE-DO construct



is always translated to

```
      Statements for Initialization
n     CONTINUE
      IF( .NOT. Condition)
        Statements f
        Computation
      GO TO n
m     CONTINUE
```

Globally, an entire flowchart can be sequentially translated from START to STOP, as long as declarations are not required. In FORTRAN, unlike other languages, the sequential translation property is strictly true until array problems are addressed. In later courses the production of programs in other high level languages is facilitated by supplying translations from the same constructs to the new languages.



A Language Processing Model

Models which are simpler than reality but which retain essential properties are widely used pedagogical tools. Absolute truth in a pedagogical model is unimportant; it is sufficient to not contradict reality. Simplicity and consistency are the required features. In this section a FORTRAN Machine is briefly described. All phenomena relevant to an introductory course can be explained in terms of the model.

The FORTRAN Machine has (initially) five Areas in which information is stored and manipulated:

1. The Variable Area holds all named data items manupulated by the program.
2. The statement Area holds executable FORTRAN statements, one per numbered line.
3. The FORMAT Area retains FORMAT statements.
4. The Input Area has 80 spaces and functions as a card buffer.
5. The Output Area has 133 spaces; a line of printed output is constructed here.

The compilation phase involves

1. storage of executable statements in the Statement Area.
2. sequential, static allocation of Variable Area spaces when variables are initially encountered.
3. storage of FORMAT statements in the FORMAT Area.
4. syntactic error checking.

An important feature of the model is a realistic picture of FORTRAN storage allocation.

The subsequent execution phase shows the interaction among the various Areas:

1. A Statement Pointer in the Statement Area moves sequentially unless altered by a GO TO statement.
2. Getting and putting values in the Variable Area shows the difference between a name and its value, the danger of uninitialized variables, and the array element referencing mechanism.
3. Simple algorithms are provided which explain how a READ or WRITE and its accompanying FORMAT interact with the Input or Output Area and associated devices. These algorithms are driven by the sequential processing of the variable lists and the specifications in the FORMAT statement.

When additional language features are introduced the FORTRAN Machine is extended, rather than defining the entire model initially.

1. Character data is described as being encoded as digit pairs; four characters will fit into a Variable Area space.
2. Subprograms require the addition of several mechanisms. Each Variable Area space is additionally named with an integer address. Spaces are allocated for function names and formal parameters during complication of subprograms. Function names and actual parameters which are not simple or subscripted variables are allocated spaces when the referencing statement is compiled. When a subprogram is referenced, the address of an actual parameter is stores in the appropriate formal parameter space. Indirect referencing is introduces to reflect the call by reference mechanism of FORTRAN. In addition, a new Subprogram Area has a space allocated for each subprogram. The Statement Area line number is saved there when a reference occurs, and RETURN uses the value to resume after the reference. Since there is only one space for a subprogram, this mechanism shows why recursion is disallowed.

This model has evolved over several semesters and seems to be an adequate vehicle for explaining in simple terms how FORTRAN language processing is accomplished. The reader may envision Machines for other languages; probably they will be more complex.

SUMMARY

We have described the objectives of our introductory course and shown some techniques and devices which help us to achieve them. Several indicators lead us to believe that these methods are successful. Students no longer remark, "I can do this, but I don't know how to start." Given a program in proper form, nearly all students will produce the same flowchart. In later courses, students learn other languages like Pl/1 and Algol quickly; they are, however, nonplussed by the lack of structure in languages like APL and SNOBOL4. It is helpful to instructors of subsequent courses that these disciplines are used when they must examine student solutions to large problems. Finally, the enrollment continues to grow in spite of its reputation

as a fairly difficult course.

Many of these concepts have been used befor [Fri76]. Two factors make this course work. The first is simplicity. A single new idea is motivated, introduced and used before another idea is presented. The simplest possible concept is used when a choice is possible.

The second factor is the constant monitoring of student behavior. The precise use of problem solving forms and FORTRAN translations is required at all times. Clerical correctness (indentation, blanks comments) is first encouraged and then required. Assignments consisting of flowchart solutions without coding encourage students to develop correct algorithms rather than using the computer in aimless debugging. Programming assignments include preliminary status reports consisting of flowcharts to discourage flowchart construction after coding. The necessity for careful motivation of students cannot be overestimated. On the instructor's part, the work load is considerably increased. But it is productive work, and the product seems to justify the effort.

References

(Con73) Conway, R. and Gries, D., An Introduction to Programming, (1973), Winthrop Publishers, Inc.

(Fri76) Friedman, F. and Koffman E., "Some Pedagogic Considerations in Teaching Elementary Programming Using Structured FORTRAN," SIGCSE Bulletin 8, 1, (February 1976), pp. 1-10.

(Gri74) Gries, D. "What Should We Teach in an Introductory Programming Course?", SIGCSE Bulletin 6, 1, (February 1974), pp. 81-89.

(Smi76) Smith, C. and Rickman, J., "Selecting Languages for Pedagogical Tools in the Computer Science Curriculum," SIGCSE Bulletin 8, 3 (September 1976), pp. 39-47.