WHAT SHOULD WE TEACH IN AN INTRODUCTORY PROGRAMMING COURSE?

David Gries
Computer Science Department
Cornell University
Ithaca, New York 14850

I. Introduction

An introductory course (and its successor) in programming should be concerned with three aspects of programming:

1. How to solve problems,
2. How to describe an algorithmic solution to a problem,
3. How to verify that an algorithm is correct.

I should like to discuss mainly the first two aspects. The third is just as important, but if the first two are carried out in a systematic fashion, the third is much easier than commonly supposed. (Note that the third step is not "debugging," because the word "debugging" conveys the impression that errors are alright -- that they are a natural phenomenon which, like flies in a house, must be found and swatted. If "debugging" was called "getting rid of one's mistakes," I'm sure most programmers would change their attitude and work harder at producing a correct program initially.)

I want to indicate what I feel is being taught as opposed to what should be taught. My main theme is, of course, structured programming, but I will discuss some other points. For example, I will outline a replacement for that crutch the computer industry has been using for so long, the flow chart. While my remarks are aimed at the instructors of introductory programming courses, I feel they are important to the whole programming profession.

Let us begin with the first aspect of programming, problem solving.

II. Problem Solving

A course in programming is unique in that it is the only course taken by a large percentage of all students, which tries to teach general problem solving methods. In history courses one often reads what Descartes, Mills, and other philosophers have to say about problem solving, but only for their philosophies; their ideas are never put into practice by the student. In English, one may come across an article on the subject, like Poe's The Philosophy of Composition [7], but it is studied for its style rather than its content.

One would assume that problem solving would be taught in a mathematics course, but it rarely is. Instead, the student is taught definitions and proofs of theorems, and he learns how to solve specific classes of problems. The bright student is one who somehow magically catches on and is able to generalize what has been taught and apply it in other situations; he has learned to solve problems. Similarly, in physics or chemistry the student learns to solve one particular kind of problem, but he does not learn general problem-solving methods.

In a programming course, we attempt to teach the student how to program anything that can be programmed -- that has an algorithmic solution. When he has finished the course, we feel he should be capable of using the computer in his mathematics course, his physics course, and even his English course. To help him we give such diverse problems as sorting, solving linear equations, drawing a graph on the line printer, "translating" English words into French, approximating π by a series, making a concordance, and so on.

In essence, we want to teach how to solve any problem by finding an algorithmic solution to it. But what do we really teach? We describe the tools the student has at his disposal (the do-loop, goto, declarations, etc.), give a few examples, and then tell him to write programs. Almost no word on how to begin, how to find ideas, how to structure his thoughts, and how to arrive at a well-structured, well-written, readable program.

Let me make an analogy to make my point clear. Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully-finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks.

You would think he was crazy! You would want instructions on designing the cabinet, his ideas on what kind of wood to use, some individual attention when you don't know what to do next, his opinion on whether you have sanded enough, and so on.

You may object that the analogy is unfair because it is impossible to teach people to think creatively. But the typical programmer rarely creates something totally original. By and large, the programs he writes can unfold in a systematic manner that is the same no matter what the subject. We should be able to teach the simple problem solving required through rules and many, many examples of their application. Unfortunately, few instructors have themselves ever thought enough how _they_ write programs in order to teach others to do it.

Fortunately, some people have thought quite hard about general problem solving, and we would all do well to read what they have written and apply it to programming. For example, Descartes presented four rules in his _Discourse on Method_ [3]:

1. Never accept anything as true unless it is certainly and evidently such: carefully avoid all precipitation and prejudement.
2. Divide each of the difficulties into as many parts as possible.
3. Think in an orderly fashion, beginning with the things which are the simplest and easiest to understand, and gradually reach towards the more complex.
4. Make enumerations so complete and reviews so general that it is certain nothing is omitted.

And Hyman and Anderson [5] give, among others, the rules

1. Run over all the elements of the problem in rapid succession, many times, until a pattern emerges which encompasses all these elements simultaneously.
2. Suspend judgement. Don't jump to conclusions. [Compare with Descartes' first rule]

You may feel these are just wise old sayings and clichés without any value, but if practiced in programming they lead to systematization and discipline, which most programmers don't have. They also can give the beginner some ideas on how to structure his maze of thoughts into something that begins to resemble an algorithm, especially if the instructors can provide him with example after example done on the blackboard, perhaps extemporaneously.

Many of you have read Dijkstra's _Notes on structured programming_ [4] which provided the impetus to the current wave of research on the programming process. Many of the ideas presented there can be seen in the light of this discussion as applications of general problem solving techniques to programming (this is not meant to detract from Dijkstra's work, which I value quite highly).

One book which I recommend most highly is Polya's _How to Solve It_ [8]. This book contains many ideas on how to help the student learn, and is valuable from this standpoint alone. The main ideas in the book are of course about problem solving, which Polya thinks of as a four-phase process (the comments in brackets are mine):

1. _Understand the problem_. [This of course should always be emphasized. Too many students are halfway through programming before they discover they don't really understand, and then have to start all over again. In programming, where the problems are often ill-defined, this phase includes precisely defining the problem.]
2. _Devise a plan_. [That is, outline the solution. This may require a good deal of work -- it often means looking at related or simpler problems, designing all the data structures, etc.]
3. _Carry out the plan_. [The plan gives the general outline; here we have to fit the pieces into a whole, which is obviously correct. Every step must be checked for correctness. In programming, this should consist of producing a top-down description of the program, irrespective of how the plan was devised.]
4. _Look back_. [We learn to program not only by programming, but by studying how we programed. Would we do it differently next time? Why was one point so hard to see? How did others solve the problem? In programming courses, various solutions handed in by students should be discussed in class, so that all can see the different viewpoints.]

In discussing problem solving as related to programming, I have not attempted to say exactly how one should teach a student how to solve problems. I can't say that -- I don't have all the answers. Conway and I have tried to introduce problem solving in our introductory text [1]. There are many things wrong with the book, and I view it as just a first attempt and not the final solution.

III. Description of algorithms

Algorithms written in programming languages are usually difficult to program, read, and understand. (Contrast this with algorithms in other languages -- recipes, knitting, etc. The only other algorithms which rival programs in complexity are those found on income tax forms!) There are several reasons for this:

1. They tend to be large -- many

are so large that one person cannot fully comprehend them.

2. Even for small programs (say under four pages), the structure of the program is often too complex for human understanding.

3. Programming languages require too much detail. Often a simple algorithm written in a suitable notation becomes quite unwieldy when translated into a programming language, only because of the extra details needed and the way the programmer implements these details.

4. Programming requires exactness and precision unknown in many other fields. Even in a mathematics paper, syntax errors and many logical errors can be understood as such and mentally corrected by the reader. But a program must be exact in every detail.

5. Algorithms change values of variables. We often try to understand an algorithm by seeing how it manipulates sample input data to produce a result. We then try to generalize to understand what happens with other input data. This is difficult to do.

The programming profession must (and has to a large extent) developed methods for overcoming these weaknesses of our programming languages. The difficulty is to get programmers to use these methods.

Point 5 can be overcome by viewing relations among the values, instead of the values themselves. I don't have time or space to discuss this here, although it is important and should be taught. See Dijkstra [4], Wirth [9], or Conway and Gries, page 192-207[1].

I would like instead to discuss three other ways to simplify the tasks of programming and understanding programs.

IIIa. The algorithmic language.

The more complicated the structure of an algorithm is (in terms of the "flow of control"), the harder the algorithm is to understand. A straight-line algorithm is the easiest to understand, while an algorithm whose various flows of control form a complete graph is probably the most difficult. Obviously, the programmer should aim for algorithms whose structure is simple — for algorithms which are "intellectually manageable," as Dijkstra [4] says. One way to do this is to allow him to use only statements which always yield a simple structure.

Let us consider an algorithmic language built up as follows. The basic or elementary statements are:

a) the assignment statement

b) input/output statements

c) procedure and macro calls (we think of a procedure or macro call as a high-level operation, performing some action. When reading the call, we are interested in what is being done, not how.)

d) English imperative statements to perform an action, such as Sort array A, Press the shirt, or Order more stock if necessary.

Note that we allow English statements (or for that matter commands in any notation which fits the problem). An algorithm is no less an algorithm just because it is not in PL/I or FORTRAN, and we should use any notation which makes an algorithm understandable. The main restriction on these basic statements is that they be understandable by themselves - out of the context in which they appear — as long as we know the definitions of variables they use. A goto is not a basic statement, because understanding it requires knowledge of the context in which it appears -- which labeled statement it branches to.

The second class of statements, the control statements, indicate the flow of control between their substatements -- they describe the possible orders of execution of their substatements. We can label the four main kinds of control statements used in programming by: sequencing, selection, iteration, and termination.

a) Sequencing. If $S_1, S_2, \ldots, S_n$ are statements, then the notation $[S_1; S_2; \ldots; S_n]$ means that first S_1 should be executed, then $S_2, \ldots,$ and finally S_n. If we want to consider the whole sequence as unit, as a simple statement, we add delimiters begin and end:

```
begin S_1;S_2;..;S_n end
```

This is called a compound statement.

b) Selection. The conventional conditional statement

```
if e then S_1 else S_2
```

allows us to choose between alternatives. A generalization of this is the case statement. Suppose that variable X contains one of n possible values 1,2,...,n, and that we want to execute one of the statements $S_1, S_2, \ldots, S_n$ depending on the value of X. We write

```
case X of
   1 : S_1;
   2 : S_2;
     :
   n : S_n end
```

Here are two examples where the notation is the same but the selection variable is not integer valued:

```
case WEATHER of
   SUNNY: Take off shirt;
  CLOUDY: Put on sweater;
   RAINY: Put on raincoat;
   SNOWY: Put on overcoat
   end
```

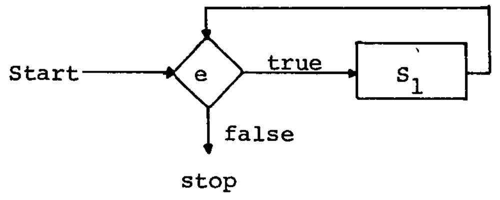
```
case  SIGN(B^2-4*A*C)  of
   -1 : X = 'complex';
    0 : X = 'double root';
   +1 : X = 'real roots'
   end
```

Breaking something up into disjoint subcases occurs in every problem-solving activity, and the student should be encouraged to use notation to fit the problem. The last step should be to put the algorithm into PL/I, FORTRAN, or whatever programming language is being used.

c) Iteration. The only loop which need be taught to express iteration is the while loop:

```
while  e  do  S1
```

whose execution can be described by the flowchart



d) Termination. The above three methods of expressing flow of control are all that is necessary to write any program. However, it is sometimes advantageous to terminate a subalgorithm named A (say) using a statement

```
terminate A
```

For example, here is one version of a binary search algorithm which looks for the value of X in the array $A_1, A_2, \ldots, A_n$ where $N \geq 1$. Upon termination $A(J) = X$ or, if X is not in A[1:N], J = 0.

```
binary search:
    begin F=1; /*if X is in list at all,*/
          L=N; /*it is in A(F:L)*/
          while F < L do
             begin J = FLOOR((F+L)/2);
                if A(J)=X then terminate
                               binary search;
                if A(J)<X then F=J+1 else
                               L=J-1;
                end end binary search
```

Programming languages already have terminate statements for subalgorithms which are procedures; they are written as RETURN.

It is easiest to teach students to write well-structured programs if we restrict the control statements they use to those which have the best chance of producing structured programs. The four control statements just described are all the introductory student needs, since they probably cover 95% or more of all programming situations.

This does not mean that programs written using these statements will be well-structured or understandable.

This does also not mean that professional programmers should only use these four control statements. A programmer must be able to develop control structures to fit his problem, when necessary. Other suitable control structures are (1) repeat S_1 until e which is equivalent to S_1; while e do S_1, (2) the statement for each a ε X do S_1 where X is a set, (3) decision tables, (4) coroutines, and (5) the conventional do loop.

One might also consider using a conventional goto, but only for a very good reason.

You may wonder why I do not include the conventional do loop as a basic tool for beginning students. First of all, as soon as they see it, they use it all the time instead of the while loop. They think in "do I = " terms, and this restricts the kinds of algorithms they can write. For example, the binary search given earlier is difficult to put in the "do I = " form. Secondly, the while loop lends itself to proving correctness of programs more easily (see the "Invariant Relation Theorem" in Dijkstra [4]). Thirdly, beginning students don't understand the conventional do loop -- they forget when the test is made and when the loop variable is incremented; and they often lose the concept of iteration and think the whole thing happens at once. If you don't believe me, try the following question on an open book midterm or final -- without hinting that such a question will be on the test. It's a simple question which any student who understands loops should be able to answer correctly in 2 minutes, especially on an open book test. It is so simple, it should be graded on an all or nothing basis. I would like to hear your results.

Rewrite the following program segment so that it accomplishes exactly the same thing, without using the "do <var> = ..." statement. You may use the while statement or gotos.

```
M = A(I);
DO J = I+1 TO N;
   IF A(J) > M THEN M = A(J);
   END;
```

IIIb. Choosing a programming language to teach. The main purpose of programming courses should be to teach general problem solving methods as applied to programming. Upon completion of, say, a two-course sequence, the student should have a good idea how to program problems in any subject matter with which he is familiar, no matter what conventional high-level language is available.

The programming language used in the course is only a vehicle allowing us to teach problem solving and the programming concepts we feel are useful (e.g. block structure, procedures, data structures). If we believe in structured programming, the language should be as close as possible to the algorithmic language just described. It should also be simple, elegant, and modular, so that features and concepts not yet taught won't get in the way.

ALGOL, ALGOL 68, ALGOL W, PASCAL, Carnegie-Mellon's BLISS, and Toronto's system language for project SUE all satisfy these conditions to a large extent. Unfortunately they are not well-known (outside the core group of computer scientists) and it would be difficult to introduce them into introductory courses. The language taught is often influenced by people outside the computer science profession, even though their opinions are not educated enough to deserve recognition. Thus the computer science departments are by and large compelled to teach FORTRAN, BASIC, or PL/I. Let us review each in turn.

FORTRAN was at the time of its creation a great step forward, and its creators are to be commended for it. However, as we look at what we want today, we find it is completely out of date. For example it has none of the control statements we feel are necessary. The compound statement, the conditional statement in its necessary generality, the iterative while loop, and the termination statement (except for procedures) are all missing. This results in programs which are usually unreadable, even if they are well-structured.

In our compiler writing course there is usually a group project consisting of implementing a small compiler. At the end of the course, I sit with each group and skim through their compiler listing to see what data structures are used, how certain statements were compiled and so on, and I can usually spot 3 or 4 errors in 15 minutes. I once made the mistake of allowing a group to write in FORTRAN, and for 30 minutes I attempted to understand their compiler with absolutely no success. This was not because of lack of knowledge; FORTRAN was my first love and I programmed it it (and assembly language) for two years.

FORTRAN is out of date and shouldn't be used unless there is absolutely nothing else available. If this is the case, use it under protest and constantly bombard the manufacturers or other authorities with complaints, suggesting they make available a more contemporary language.

BASIC is a FORTRAN-like language with even severer restrictions. For example, the name of a variable must be a letter followed optionally by one digit. It does not enjoy FORTRAN's reputation for being the first high-level language, and should never have come into existence. When it was contemplated, its designers should have done their research to see what programming and programming languages are all about before plunging in.

I hesitate to think of the thousands of students learning to think in an ad hoc, unreadable language, under the guise of time-sharing. I have doubts about teaching students to think "on-line"; algorithms should be designed and written slowly and quietly at one's desk. Only when assured of correctness is it time to go to the computer and test the algorithm on-line.

If one must program in a well-known language, the only choice is PL/I. It has the compound statement, the conditional statement and the while loop. The case statement must of course be written using goto's and perhaps label variables, but this is all right as long as it is made clear that the case statement is being simulated. Similarly the termination statement can be done with a goto. But we don't go to do anything else, we go to stop execution of the algorithm. For example, the binary search algorithm given earlier would be written as

```
/*binary search for X in A(1:N), upon ter-
  mination if X was in the list A(J) = X,
  otherwise J = 0 */
  /*Invariant relation of loop:  If X is
    in A(1:N), then it is in A(F:L)*/
    F = 1; L = N;
    DO WHILE (F  <= L);
        J = FLOOR((F+L)/2);
        IF A(J) = X THEN GOTO
                    TERMINATEBINSEARCH;
        IF A(J) < X THEN F = J + 1;
                    ELSE L = J - 1;
        END;
    J = 0;
    TERMINATEBINSEARCH:;
```

I choose PL/I solely because it's the best choice, not because I particularly like it. In fact, it took a lot of thinking before I decided to write an introductory programming book based on PL/I [1]. What's wrong with PL/I? Its syntax is enough to offend anyone who has studied English grammar; its data structure facilities (structures) could have been less clumsy and much more elegant and usable (e.g. more like PASCAL, where one defines new data types); it is not modular, as claimed (it is difficult to teach one feature without another getting in the way); its astonishment factor is much too high (e.g. what is 25 + 1/3 ?); its parallel programming features are difficult to understand and use correctly; and so on.

Yet, if we stick to its simpler features, the language is usable. In our first programming course, while teaching how to program, the students learn about (1) simple variables and arrays; (2) FIXED DECIMAL, FLOAT DECIMAL, and CHARACTER data types; (3) expressions and assignment statements using these data types; (4) GET LIST and PUT LIST; (5) the compound statement, conditional statement, while loop, and goto as used to simulate the termination statement; and (6) external procedures with parameters and STATIC variables (but not EXTERNAL variables). Primarily, the course is aimed at showing how to design and program well-structured programs.

IIIc. Program documentation. The flow chart has always been used as the main tool for describing programs. Indeed, it is often required as part of program documentation. When used correctly, it can be of help during the programming process and can certainly aid in understanding.

Unfortunately, flow charts have several disadvantages which are severe enough to cause us to stop using them. Of course, there will still be occasion to use them, but for the most part high level programs should be documented using the method outlined below.

What's wrong with flow charts? First of all, I have rarely seen a programmer who liked to draw them. Consequently, the flow chart is often drawn after the program is written and debugged; in fact there are systems which will draw a flow chart from the finished program. Used in this way, the flow chart is of no use to the person who needs it the most, the programmer. Program documentation should be written while the program is being written, if not before, and should be used by the programmer in proving correctness and in checking his program out.

Secondly, the flow chart allows, even encourages complicated program structure. Given a flow chart, it is always easier to correct mistakes or add to the chart by drawing in a few more arrows and boxes, than it is to restructure the chart to make it simpler and more systematic. And if the flow chart is drawn from a completed program, then it can be no better in structure than that program; its use will be solely to provide a two-dimensional representation.

We need program documentation which encourages systematic, structured programming, and which can be used by the programmer as he programs. Furthermore, the rules which govern its use must be such as to force the programmer into trying to get a correct program before he begins testing.

The method I advocate is a top-down description of the program, using indentation to indicate refinement.

Now I don't expect every programmer to program using step-wise refinement. Problem solving is an individual, personal thing, and although we can give rules to aid in understanding problem solving, we cannot expect all ideas to emerge in a rigid, disciplined fashion. However on the ideas for a program have been generated, the programmer must produce a top-down description of his program.

Polya's second and third phases of problem solving are: devising a plan and carrying out the plan. In programming terms this means

Devise a plan: Get the ideas for the program; lay out the general structure, play with various subparts so that data structures can be determined, etc.

Carry out the plan: Systematically produce a top-down description of the program, checking each refinement for correctness.

For those of you who are not familiar with the terms "top-down" and "step-wise refinement," let me briefly explain.

Top-down programming, or step-wise refinement, is a method of producing a program which proceeds as follows. One begins with the statement S1: "solve the problem." One then refines this by giving a sequence of statements [S1.1, S1.2, S1.3, ...,S1.n] which solve that problem. S1 specifies what to do; the sequence [S1.1, ...,S1.n] indicates how to do it. The sequence is an algorithm which, if executed, produces the desired result. Each statement S1.i is a command written in English, PL/I or any suitable notation. For example, S1.2 might be a loop while e do S1.2.1, where S1.2.1 is another command in English, Pl/I or any suitable notation.

The main point is that the sequence [S1.1,...,S1.n] is specified at a high enough level that its correctness is obvious. This means that a lot of detail has yet to be uncovered.

One then proceeds to refine each of the statements S1.1,...,S1.n in turn. Since these are independent of each other in the sense that each can be understood out of context, the order in which these are refined is immaterial.

This process continues until all the refinements have led to statements in the programming language. Each refinement is small enough so that its correctness is obvious, and thus the whole program must be correct.

One also makes data structure refinements when necessary. For example one might decide to implement a set by an array, a linked list, or a bit string, depending on the operations performed on the set. This would require suitable refinements or changes in statements which operate on that set.

It is not implied that a program can be generated purely in this fashion. One makes mistakes which must be corrected; one may see a better way to implement something that requires one to "back up" and redo part of the program, and so on.

A top-down description of a program consists of the program itself, together with the series of refinements which led to it. Thus, the reader is shown S1, then the sequence [S1.1,...,S1.n], then the refinement of these, and so on. From this, he can easily deduce the correctness of the program, since each refinement is small enough to easily understand.

The complete top-down description can be interspersed within the program, using indentation to show refinement. The statements not in PL/I are written as comments, as indicated below:

```
/*S1*/
     S1.1;
     S1.2;
     /*S1.3*/
          IF e
          THEN S1.3.1
          ELSE S1.3.2;
     /*S1.4*/
          S1.4.1;
          DO WHILE (B);
               S1.4.2
               END;
          S1.4.3
```

The advantage of this scheme is that the reader can understand the program at any level of detail he desires. He can read just S1. To understand how S1 is implemented, he reads its refinement -- he indents his thinking 5 columns and reads the sequence [S1.1, S1.2, S1.3, S1.4]. If he has to, he can then easily understand how S1.4 is implemented by reading the sequence of three statements which constitute its refinement.

A second advantage of this scheme is that it forces the programmer to put his program together in a systematic, well-structured manner. If he is careful, correctness is obvious, and the only errors that testing should find are misprints, keypunch errors, and so on.

You may complain that there is too much indentation to worry about, and that for large programs the final statements will be so far indented that they can only use, say, columns 60 - 72! But no program segment should be over 1 or at most 2 pages long. Longer than this becomes too hard to understand anyway. Liberal use should be made of macros and procedures to keep each program segment under 1 or 2 pages. There will be of course exceptions to this rule, but for the most part it should be followed.

Please don't take this as a complete discussion of how I feel programs should be described; this is just an outline. We are teaching this in our introductory courses at Cornell, and by and large it works well. Programs have much better structure to them than before, and it is much easier to look at various parts and ascertain their correctness. This scheme also tells students, perhaps for the first time, what comments are for and where they come from. Previously, comments were something to insert after the program was finished, and it wasn't clear what kinds of comments to write. Now, a comment is a command telling what to do, its indented refinement explains how to do it.

Of course there are problems. Students sometimes tend to go overboard. While the program below follows our rules, each comment and its refinement are the same, and the comments can be omitted. The second segment below is less redundant and clearer.

```
/*Initialize I*/
     I = 0;
/*Read in command*/
     GET LIST(COMMAND);
/*Loop until command is 'ENDRUN'*/
     DO WHILE(COMMAND≠'ENDRUN');
          Process command
          /*Read in command*/
               GET LIST(COMMAND);
          END;


I = 0;
GET LIST(COMMAND);
DO WHILE(COMMAND≠'ENDRUN')'
     Process command;
     GET LIST(COMMAND);
     END;
```

One of the hardest tasks for an instructor is to grade students' programs. It is not enough to grade solely on the correct result being printed out. More important is to look painstakingly through the program, writing comments to the student about what was done correctly, indicating where something could have been done more systematically, and pointing out where it is obvious that the program is not well-structured, etc. Let me illustrate what I mean with a live example from a course I am currently teaching.

The 3 - 4 page program from which the segments below were taken did simplified inventory accounting for a warehouse. Based on a sequence of commands entered through the normal input file, it kept track of the number of units of each item in stock, printed out sales slips, ordered when stocks got too low, and so forth.

```
DECLARE NO_TO_ORDER FIXED DECIMAL INITIAL(0);
     :
     /*Determine how many to order and
       print order*/
          IF X < Y THEN NO_TO_ORDER = X;
          IF Z < W THEN NO_TO_ORDER =
            NO_TO_ORDER + Z;
          :
          Print out order;
          :
          NO_TO_ORDER = 0;
```

I tried to understand the refinement of the statement "Determine how many to order and print order" (the names X, Y, Z, W were originally longer and mnemonic; I have changed them since they are immaterial to this discussion). My first thought when reading the first conditional statement was that NO_TO_ORDER is not initialized if X $\geq$ Y. Thus I had to see if NO_TO_ORDER was initialized earlier, or if X $<$ Y always held at this point of execution. I finally found the initialization in the declaration.

But then I began to think about the next time this subalgorithm would be executed -- what value would NO_TO_ORDER have?

After a minute or two, I finally found the statement NO_TO_ORDER=0 at the end of the subalgorithm.

I had to spend several extra minutes understanding the program because the program was not well-structured. Even though the program worked, the student lost some points; in return he received a written explanation from which I hoped he would learn something.

The program was not well-structured because that last statement of the refinement, NO_TO_ORDER = 0;, had nothing to do with "Determine how many to order and print order." True, it sets NO_TO_ORDER up for the next execution of the algorithm, but this was not stated as a purpose of the algorithm. I would have had no trouble understanding if the student had written the equally efficient segment

```
/*Determine how many to order and print
  order*/
    NO_TO_ORDER = 0;
    IF X < Y THEN NO_TO_ORDER =
      NO_TO_ORDER + X;
    IF Z < W THEN NO_TO_ORDER =
      NO_TO_ORDER + Z;
    :
    Print out order;
```

Of course it is difficult to put so much time and effort into grading, but in the large programming courses we teach these days, it is the only source of individual contact between instructor and student. The student can learn the basic programming language tools by himself but he needs individual help with problem solving, and with understanding how and why we want the program written. It is not enough to mark on his program "-5 because you didn't follow rule such and such." We must also explain how in this particular instance violation of the rule led to a less-understandable program or even to an undetected error in the program.

I would like to teach programming to a group of, say, 15 students (rather than 150) just to see whether the extra individual attention would produce better programmers. I'm sure it would.

IV. Conclusions

The programming profession and computer industry are currently in a "software crisis," brought on by the fact that the profession is constantly asked to solve larger and more complex problems than we ever dreamed of. The old programming techniques don't work on the larger problems. The solution is to educate new programmers in a different manner, and to re-educate the old programmers. The emphasis should be on systematic programming based on tested problem solving principles, on discipline and carefulness, and on the production of neat, elegant, simple algorithms which are proved correct before they are tested on a computer. Note that elegance and simplicity do not preclude efficiency. On the contrary, only through simplicity can we see a way to make things more efficient. I would rather have a correct algorithm which runs in ten minutes than one which runs in one minute but whose correctness I cannot ascertain.

However it is not enough to stand in front of the class and mouth the clichés of problem solving and structured programming. The students will easily sense whether you believe in what you tell them, and whether you yourself practice what you teach. To teach structured programming you must practice it yourself. Unfortunately it takes time and hard work to switch from bad habits to better ways of programming (just as it takes time and effort to play golf correctly). Anybody contemplating teaching programming has a lot of preparation to do. He should study Polya's book How to Solve It [8], and then read what others have to say on problem solving. The book by Dahl, Dijkstra and Hoare [4] should be read, especially Dijkstra's Notes on Structured Programming. Also skim through Wirth's book on Systematic Programming [9], and An Introduction to Programming by Conway and myself [1] to see how we approach the subject. Finally, and this is important, write several programs, both large and small, using the tools and techniques advocated. I'm sure if you do this you will be pleasantly surprised.

I am not at all sure that any drastic change will take place in the Universities within the next few years, although I hope I'm wrong. You would think that the University, where one searches for truth and knowledge, would be the place for innovative thinking, for people who are tuned to new and better ideas. Yet Daniel McCracken made a survey of 40 representative universities throughout the country about one year ago [6], with which he concluded that

"Nobody would claim that FORTRAN is ideal for anything, from teachability, to understandability of finished programs, to extensibility. Yet it is being used by a whopping 70% of the students covered by the survey, and the consensus among the university people who responded to the survey is that nothing is going to change much anytime soon."

Does this sound like educators who are committed to teaching concepts, to teaching people what they need to know to prepare for the future?

Let's get with it and find what programming is all about, and then make a concerted effort to teach a better style of programming to our students.

V. References

[1] Conway, R. and D. Gries. An Introduction to Programming, Winthrop Pub., Cambridge, Mass., 1973.

[2] Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare. Structured Programming. Academic Press, New York, 1972.

[3] Descartes, René. Discourse on Method, 1637. Can be found in numerous texts and anthologies in philosophy.

[4] Dijkstra, E.W. Notes on Structured Programming. In [2]. This provided the impetus to the current wave of research and articles on the programming process and the structure of programs. Unfortunately, the term "structured programming" is not defined in it! Consequently, different people have different ideas on what the term means. It deserves to be read thoroughly by every programmer.

[5] Hyman, R. and B. Anderson. Solving Problems. International Science and Technology (Sept. 1965), 36-41.

[6] McCracken, D. Is there a FORTRAN in your future? Datamation (May 1973), 236-237.

[7] Poe, Edgar Allen. The Philosophy of Composition. Graham's Magazine, April 1846. (Appears in Stearn (editor), The Portable Poe, Viking Press, 1945. This amazing article describes how Poe wrote The Raven, "step by step, to its completion with the precision and rigid consequence of a mathematical problem." We would call it top-down programming.

[8] Polya, G. How to Solve It. Princeton University Press, Princeton, N.J., 1945. Every instructor in an introductory programming course should read this book.

[9] Wirth, N. Systematic Programming: An Introduction. Prentice-Hall, Englewood Cliffs, N.J., 1973.