EDUCATING THE PROGRAMMER:

NOTATION, PROOFS AND THE DEVELOPMENT OF PROGRAMS+

David Gries

TR 80-414

Department of Computer Science
Cornell University
Ithaca, New York 14853

+Work supported by NSF grant MCS76-22360.

Educating the Programmer: Notation, Proofs and the Development of Programs+

David Gries
Computer Science, Cornell University
Ithaca, New York, USA

The current state of affairs in programming is discussed. The opinion is expressed that effective programming requires more "mathematical maturity" than most programmers have. Further, education in formal logic, which is used (often informally) to reason about programs and specifications, and in a theory of programming could do much to increase the programmer's competence. Such education could lead to programming becoming more of a science than just an art. Examples are given throughout to support the opinions presented.

1. INTRODUCTION: VIEWS OF THE 70's++

The 1970's saw what might seem a remarkable increase in our knowledge of programming. It was just in 1968 that a NATO conference on "software engineering" was convened [1], in which academicians and industrialists discussed the software crisis and admitted that how to program -- and how to teach it -- was really a mystery. Recognition of this crisis stimulated much research. Consequently, the 1970's saw the emergence of "structured programming", which most people profess to practice, of the idea of program correctness and how correctness might be proven, of management techniques such as the "chief programmer team", of documentation aids and devices that attempt to force the programmer to document well (e.g. structured flaw charts, HIPO diagrams), and of new programming notations (e.g. Pascal and, of late, the U.S. Department of Defense notation Ada).

Today, people in industry tell me that young programmers are much better trained than their counterparts of ten years ago, and that their programs are in general much more readable (chiefly because they are no longer piles of spaghetti!). This has made people generally happier with the state of affairs than they were 10 years ago.

But there is also a pessimistic side of the coin. In general, the field still lacks "professionalism"; programming is still too much of a sloppy art instead of a science, and relatively few programmers seem interested in understanding programming enough to really study it. This saddens me somewhat, for I think that enough theoretical foundations have been laid in the past ten years to have made programming more of an orderly, professional process.

Let me explain a bit more why I feel this way. Many years ago, Tony Hoare called _structured programming_ the task of organizing one's thoughts in a way that leads, in a reasonable amount of time, to an understandable expression of a computing task. While this definition gives no insight into _how_ structured programming can be performed, it does succinctly state _what_ we should expect from it, and thus allows us to determine whether programmers practice it. The answer is no, for three reasons. First, the task is still not completed in a reasonable amount of time, in spite of all our advances, as is evidenced by the frequent cost overruns and missed deadlines. Secondly, the final program is rarely as readable and understandable as it could be (although progress has been made). Thirdly, programs rarely satisfy their specifications; they are replete with errors, some of which are not found for years (if ever).

Perhaps this is why over 70% of all programming is now maintenance of old programs (which means that only 30% is devoted to developing new ones), and why each year the percentage increases: we don't know what we are doing.

Here is a specific example to illustrate the general lack of proficiency. In 1976 I had to write an algorithm to "right-justify" a line of

---
+Work supported by NSF grant MCS76-22360.
++The words he, his and him are used to denote someone of either sex.

text -- to insert blanks in a given manner between words so that the last word would end in a certain column. I applied what I thought were effective techniques (using correctness-proof ideas, etc.). The result was nice enough for me to consider writing it for others to read, and referees thought enough of it to honor it with publication [2].

As an experiment, I asked 40-50 graduate students, faculty members and programmers from industry to solve the problem. They were explicitly told that this was an experiment, that the first concern was correctness and clarity, and that the second concern was efficiency. Nevertheless, over half of them wrote incorrect programs! Moreover, my program was slightly more efficient than all the others; due to the style of programming, I had developed a different solution. The errors in many of the programs were minor, but of such a nature that testing would probably not uncover them. I also took no longer to create my correct program than they did to create their incorrect ones, although it did take me a long time to describe the development in a technical report.

It is this general lack of proficiency at writing even small programs that causes me concern.

In my country, the USA, we now have over 105 000 undergraduates between the ages of 18 and 21 studying computer science as a major topic. That is an enormous number -- what a chance to make a lasting impression, to produce young people who will enter industry and effect a change! Unfortunately, while most get some introduction to the use of simple control structures, too many are having their first algorithmic thoughts polluted by BASIC and FORTRAN, and too few are being exposed to the solid theoretical background that I feel is so important.

Even at the graduate level the wrong viewpoint is often taught. I recently talked with a young Ph.D. who had just finished a thesis in program verification. I asked how many students majored in that area in his department. He replied that very few did, because it was so hard. One had to attach assertions to flaw charts, with no insight into how or why it was done, one had to study esoteric kinds of logic, with emphasis on models and consistency, and so forth, and it was just too difficult. I asked whether in any course the idea of producing a program and its correctness-proof hand-in-hand, with the latter actually leading the way, was taught. No, it wasn't, and the idea had not even occurred to him! And he was intrigued at some of the program developments I showed him.

Here was a computer scientist who had spent 3 or 4 years studying verification and had not even heard of some important reasons for and principles behind his work. I view a proof of correctness as my obligation when developing a program, and as something to guide me in the development, but certainly not something to be studied in and for itself. I agree with DeMillo, Lipton and Perlis in their article Social processes and proofs of theorems and programs [3], if they are talking about mindless verification.

But enough! I have been talking about what is wrong with the field in order to make clear to you that all is not as it should be. It is time to stop complaining and begin talking about how it could be.

2. CAN PROGRAMMING BECOME A PROFESSION?

To me, programming is a difficult intellectual challenge, which requires a good deal of mathematical maturity. As with mathematical theorems, it is the programmer's duty to prove his program correct. A theorem, or a program, is a tool for others to use, but in order to use it one must be assured of its correctness.

Perhaps I should explain what I mean by a proof, for some people are frightened by the word. According to Webster's Third New International Dictionary, a proof is "the cogency of evidence that compels acceptance by the mind of a truth or a fact". Thus a proof is an argument that convinces the reader of the truth of a statement. The definition says nothing about the language in which the proof is written or about how formal it must be.

In this sense, every programmer tries to "prove" his program correct, for he tries to come up with arguments that convince himself of its correctness. That he is often not up to the task is evident from the fact that his programs are replete with errors even after much testing.

Ten years ago, we did not yet have the technical understanding to know what a proof of program correctness should entail; we were just groping our way towards that understanding. So to require proofs of any kind at that time would have been silly.

We have, however, come a long way since then; we have established much of the necessary

theoretical foundations. Hence, I am actually optimistic about the future. We are reaching a stage where we can begin to talk of the science, rather than the art of programming, where these two terms are used in the sense given in Fowler and Gowers's Dictionary of Modern English Usage:

> the term science is extended to denote a department of practical work that depends on the knowledge and conscious application of principles; an art, on the other hand, being understood to require merely knowledge of traditional rules and skill acquired by habit.

This does not mean that programming will become a sterile, simple task. There will always be creativity and excitement in it, and it will always require skill and intellectual effort. But the emergence of principles based on theoretical foundations means that we can do a better job on the programming task and, more importantly, we can teach others how to do it better.

In this talk I am arguing for better understanding and education in what has been called programming "in-the-small", as opposed to programming "in-the-large", a term coined by Frank DeRemer and Hans Kron in 1975. One reason for this is that I do think the methods will "scale up" to large systems of programs, but there is another reason. To put it quite simply, I believe that one cannot, simply cannot expect to develop large programs or systems of programs effectively until one can develop small programs effectively.

This argument was put forward by Edsger W. Dijkstra in an extremely convincing manner: Suppose a program consists of n small components, each with a probability p of being correct. Then the probability P that the whole program is correct certainly satisfies $P < p^n$. Since n is large in any good-sized program, to have any hope that the program is correct requires p to be very, very close to 1.

Please stop for a moment and read that argument again. It has convinced me to spend a good deal of time in the past ten years studying programming in-the-small. Others seem capable of brushing the implications of this argument aside, but I can't.

Of course, other kinds of education and research are necessary as well. I can think of a number of areas in which research and education would help, e.g.

(1) Programming notations, or languages (e.g. Ada),
(2) Mechanical aids, e.g. debugging tools, verifiers,
(3) Management techniques.

But the general lack of programming proficiency (in myself as well as others) forces me to conclude that the best way to make significant progress is to increase technical competence, and at first "in-the-small". Note that items (1)-(3) yield only supplemental tools, which aid in the use of the mental tools of the programmer, and no supplemental tool can make up entirely for lack of mental skill.

Another way to state the case is to say that increasing technical competence is preventive medicine for reducing the number of unhealthy programs. The cause of errors in programs is, to some extent, incompetence, and increasing competence should help prevent errors. Tools such as debugging aids, on the other hand, are aimed at curing sickness after it has spread. While both prevention and cure are necessary, in the long run prevention is better.

I have thus far given you my vague opinion on what is needed: more and better education to increase technical competence. Let me now be more specific. In the succeeding sections, I will discuss the following, in turn.

(1) Predicate calculus
(2) A theory of programming
(3) Searching for simplicity
(4) Using suitable notations

3. TEACHING THE PREDICATE CALCULUS

Programmers constantly deal with propositional statements about program variables. They must see logical connections, must deduce conclusions from hypotheses, and so on. Yet, rarely are they taught the fundamentals of such reasoning; they are not given the necessary mental tools. They usually learn about Boolean expressions, but they don't learn how to reason about them.

Every programmer needs a basic course in formal logic (in the propositional calculus and predicate calculus), for these form the foundations for our reasoning, be it formal or informal.

Unfortunately, logic is often taught from the logician's point of view, and not the programmer's. The logician is more interested in logic for its own sake, while the programmer

is interested only in using logic as a practical, everyday tool. For example, he needs to understand how a specification can describe a set of states; he must have facility at translating (usually ambiguous) descriptions of what a program is to do into a formal specification written in the predicate calculus; he should be an expert at manipulating logical formulae to prove conclusions from hypotheses. One doesn't need much deep logic; one needs to learn some basic material extremely well.

A sound knowledge of predicate calculus is necessary especially in order to understand the theoretical foundations to be explained later. For example, a basic idea in one approach to programming is that of a "weakest precondition" wp(S, R) for a statement S with respect to predicate R: a predicate describing the largest set of states such that execution of S begun in any one of those states is guaranteed to terminate in a state satisfying R. To fully understand wp requires knowledge of how predicates describe sets of states. For example, using this idea the assignment statement is defined as

$$wp(x := e, R) = R^x_e$$

where the predicate to the right of = is derived by replacing all free occurrences of x in R by e. For example, wp("x:= x+1", x > 0) = x+1 > 0. This rule yields a mathematical understanding of assignment and <u>can</u> be practically applied. But unless one thoroughly understands textual substitution and the relation between a predicate and the set of states it represents, this simple proof rule cannot be fully appreciated or used.

Here is an example of the kind of reasoning that programmers should be able to perform easily. It is not taken from a programming domain, but from the game <u>WFF'N PROOF</u> -- <u>the game of MODERN LOGIC</u>.

<u>The tardy bus problem</u>. Suppose the following three statements are given:

(1) If Bill takes the bus, then Bill misses his appointment, if the bus is late.
(2) Bill shouldn't go home, if (a) Bill misses his appointment, and (b) Bill feels downcast.
(3) If Bill doesn't get the job, then (a) Bill feels downcast, and (b) Bill should go home.

Assuming these statements are true, which of the statements are valid? (Give proofs of the valid ones and counterexamples for the invalid ones.)

(1) If the bus is late, then (a) Bill doesn't take the bus, or Bill doesn't miss his appointment, if (b) Bill doesn't get the job.
(2) If Bill does get the job, then (a) Bill doesn't feel downcast, or (b) Bill shouldn't go home.
(3) If (a) Bill should go home, and Bill takes the bus, then (b) Bill doesn't feel downcast, if the bus is late.

Part of the difficulty in solving such puzzles is the English: it is difficult to parse some of the sentences. Programmers have the same problem in understanding English specifications and translating them into a more formal form. Moreover, once understood, one needs to know how to effectively reason about the specification -- how to prove conclusions from hypotheses.

Unfortunately, the computer science "Curriculum 78" [4] does not include a course on logic, although 7 other courses in mathematics are included because of their "relevance" to computer science. I would hope that the next revision would be more up to date.

4. TEACHING A THEORY OF PROGRAMMING

When it comes to the programming activity itself, I believe the emphasis should be on constructing programs that, with a very high degree of probability, are correct. I think this can be done, if we excuse the typical kinds of syntactic errors we tend to make -- after all, we are only human. But it requires knowledge of and experience with principles of program development that are based on developing a program and its proof hand-in-hand. This requires more mathematical maturity than many programmers currently possess.

My own tastes run to Edsger W. Dijkstra's way of thinking of programming, and in my opinion his book [5] represents a significant advance in our understanding of programming. I will try to illustrate two of his major points here, in the hope that you will be convinced that there is something to this "disciplined" approach to programming. But please remember that I can only illustrate a few basic points, using very small examples. Many people, when seeing similar introductions, are skeptical and say, "I see how it works on that example, but does it generalize?" Or they question whether

the material is too difficult for the "average programmer" to learn.

To the first question, I must reply yes. More to the point, study and practice of the method yields a new outlook on the programming process, which increases understanding and proficiency in ways that cannot be quantified. My teaching has also been influenced; I feel I have a better grasp of fundamentals, and the theory gives me reasons for introducing certain concepts and ideas, even though I do not (yet) teach all the formalism in an introductory course.

To the second question, I must answer that my job is to investigate the process of programming, and if my investigations point to the fact that certain technical material is necessary, that programming is inherently difficult, I am forced to tell you that. You may decide to reject my technical ideas based on political or social considerations, but please be sure that you realize the nature of your considerations.

4.1 "Weakest preconditions and thinking backwards"

As mentioned earlier, one of the key points in Dijkstra's work is the notion of a weakest precondition wp(S, R). Previously, the notation {P} S {R} had been used as a specification for program S, with the meaning: if execution of S begins in a state satisfying predicate P, then execution will terminate in a state satisfying R. We see now that this notation is equivalent to $P \Rightarrow wp(S, R)$.

Given a specification {P} S {R} and asked to develop S, the definition of wp influences us to attempt to develop S based mainly on R, and at the same time to develop wp(S, R); when finished, we have to prove that $P \Rightarrow wp(S, R)$.

At first, this seems backward; our operational habits make us think we should develop a program based on the precondition, because the program is executed from beginning to end. But programming is a "goal-oriented" activity, with much more emphasis on the postcondition R than the precondition P. An example will illustrate this.

Consider writing a program (segment) to store the maximum of two variables x and y in a variable z. Thus, we want to write a program S satisfying {P: true} S {R: z = max(x, y)}. The postcondition can be put in the form

$$R: z \geq x \wedge z \geq y \wedge (z = x \vee z = y).$$

We attempt to derive S solely from R: how can we assign to z to establish R? We can establish the part $z = x$ of R by executing $z := x$, but this will establish R only under certain conditions. To determine those conditions we derive

$$\begin{aligned} wp(\text{"}z := x\text{"}, R) &= R^z_x \\ &\equiv x \geq x \wedge x \geq y \wedge (x = x \vee x = y) \\ &\equiv x \geq y. \end{aligned}$$

Hence execution of $z := x$ establishes R if and only if initially $x \geq y$. Similarly, we see that execution of $z := y$ establishes R if and only if $y \geq x$. This leads us to construct the following alternative statement in Dijkstra's notation:

```
if x ≥ y → z:= x
 □ y ≥ x → z:= y
fi
```

Both formal and informal reasoning allows us to conclude that execution will always establish R, so that the precondition is the desired one: true.

Given R, we developed a statement S, and then found that the desired precondition true was satisfactory. The reader is now invited to do the opposite:

> Develop S from the precondition P (= true) only;
> When finished, prove that execution of S with P true establishes R.

The possibility of arriving at a correct statement S in this manner is obviously remote.

This idea that programming is a goal-oriented activity needs much more explanation than I can give in this short talk, but I hope that I have convinced you that there is something to it.

4.2 Loop invariants

Another key point in this discipline of programming concerns invariants of loops. (The notion of an invariant was also crucial in earlier ground-breaking work by Bob Floyd in 1967 and Tony Hoare in 1969.)

Let us consider a loop **do** B → S **od**. I am using Dijkstra's notation, because, at this point, it seems to me to be the most simple, concise and flexible. This loop is equivalent to the PL/I loop DO WHILE (B); S END;. The following theorem has been proved about this

loop.

__Theorem__. Let P be a predicate that satisfies
$P \Rightarrow wp(S, P)$.
Let t be an integer function of the program variables that satisfies
$(P \wedge B) \Rightarrow t > 0$ and
$(P \wedge B) \Rightarrow wp(\text{"}T{:=}\ t;\ S\text{"}, t < T)$
where T is a new program variable. Then
$P \Rightarrow wp(\text{"}\mathbf{do}\ B \rightarrow S\ \mathbf{od}\text{"}, P \wedge \neg B)$.

The first hypothesis, $P \Rightarrow wp(S, P)$, states that execution of the loop body leaves P invariantly true. Hence P is called an __invariant relation__ of the loop. The second two hypotheses define properties of an integer function t, which can be viewed as an upper bound on the number of iterations still to be performed. Thus these indicate that execution of the loop will terminate. One of these says that as long as another iteration is necessary the bound on the number of iterations still to be performed is greater than zero; the other says that execution of the loop body decreases the bound by at least 1.

Does this seem difficult? It may, at first. Nevertheless, it is one of the most exciting developments in programming, for it has taught us how to understand loops, which are one of the most difficult parts of programming. Moreover, it has provided a way to teach others to understand loops. The effect of such formal developments can not be overestimated!

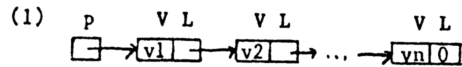
To use the theroem, one must have a suitable loop invariant. Furthermore, it is wise to develop the invariant __before__ writing the loop (or in parallel with writing the loop, through a trial and error process), as the following example shows.

4.3 __An example of the development of a loop__

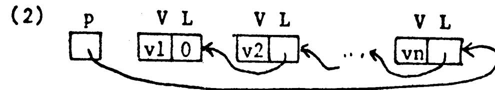
Consider the problem of reversing a linked list. Two arrays V[1:100] (for __V__alue) and L[1:100] (for __L__ink) and a variable p are used to implement a linked list of values (v1, ..., vn) for some n, $0 \le n \le 100$:

$$v1 = V[p],\ v2 = V[L[p]],$$
$$v3 = V[L[L[p]]] = V[L^2[p]],\ \ldots,$$
$$vn = V[L^{n-1}[p]],\ L^n[p] = 0.$$

In pictures, we have

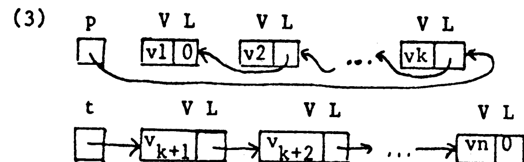
(1)


It is desired to reverse the linked list, so that p represents the list (vn, ..., v1). Thus, upon termination, we should have

(2)


No other arrays may be used. Array V should __not__ be changed; only array L and variable p may be changed.

It looks like a loop is necessary, since there is no limit on the length of the list. Furthermore, it seems reasonable to have a loop that at each iteration changes one of the L fields. Initially, the invariant must "look like" the initial conditions, and upon termination it must look like the final result. Thus, we seek an invariant of which both the initial condition and result are instances. With this in mind, it is fairly easy to represent the invariant by an informal picture:

(3)


Thus, p represents the part of the list that has been reversed; t the part that has not. It is then easy to write down the algorithm:

```
t:= p; p:= 0;  {Invariant has been esta-
                blished}
do t ≠ 0 → p, t, L[t]:= t, L[t], p od
```

The termination fuction t of the theorem is the number of elements on the list represented by variable t.

Isn't that a neat little program? I have seen programmers slave over this small, trivial problem for over a half hour, and still not know whether their result was correct. Writing down an invariant __first__, in order to crystalize the idea behind the loop, leads almost immediately to the program.

I must confess to two errors concerning the development of this algorithm. First, most languages do not have the multiple assignment statement used above. Hence, I must "sequentialize" -- write the multiple assignment as a sequence of simple assignments -- and, invariably, I make a mistake when doing this. The multiple assignment is so basic to my way of thinking -- it represents a simple change of

state by changing a set of variables rather than just one -- that I think it should be included in all programming languages. If not included, it should at least be taught as a concept in programming courses.

Secondly, the use of pictures, as above, is nice, because it lets us see what the idea is more quickly than a complicated predicate calculus statement would. But it can <u>too easily</u> lead to errors or inefficiencies, for it is just too difficult to convey everything in a picture because it looks like a single example. To illustrate, my first attempt at writing the loop from the picture was

```
if p ≠ 0 → t:= L[p]; L[p]:= 0;
           do t ≠ 0 → p, t, L[p]:= t, L[t], p od
 □ p = 0 → skip
fi
```

This was because the picture of the invariant lent no clue as to the case $p = 0$ -- when p represents an empty list -- and I was led to think that p should have at least one element in it. Gary Levin, a graduate student at Cornell, showed me my error. Let $t\rightarrow$ represent the list of values in a linked list whose head pointer is t, and let $t\leftarrow$ represent the same list of values but reversed. Then the input specification can be written as $p\rightarrow = (v1, \ldots, vn)$ and the desired result can be written as $p\leftarrow = (v1, \ldots, vn)$. It is then fairly easy to see that a good invariant is

$$p\leftarrow \circ\, t\rightarrow = (v1, \ldots, vn).$$

where o denotes concatenation of lists. From this it is easy to see that "t:= p; p:= 0" is a valid initialization, and that the case $p = 0$ does not have to be handled separately.

4.4 <u>A second example</u>

Let us consider another example, which shows nicely how research in programming has taught us to reason about iterative processes. Consider an urn filled with a number of black balls and white balls. There are also enough balls outside the urn to play the following game. We want to reduce the number of balls in the urn to 1 by repeating the following process as often as necessary.

> Pick any two balls in the urn. If both are the same color then throw them away, but put another black ball into the urn; if they are different colors then return the white one to the urn and throw the black one away.

Each "execution" of the above process reduces the number of balls in the urn by 1; when only one ball is left, the game is over. Now, what, if anything, can be said about the color of the final ball in the urn in relation to the original number of black balls and white balls?

Right now, you are probably thinking about the problem in terms of test cases: what happens if initially there are two white balls? Two black balls? One and one? Two and one? Note that such thinking is akin to programming by test cases: based on some test cases write a program; look for more test cases and revise the program, etc. Experience has shown that this process simply doesn't work well. First, one soon generates so many cases that confusion results. Secondly, continually patching a program to handle new cases results in a mess.

Let us try to be more effective in finding a solution to this. Note that the game is an iterative process, and there should be some property that holds about the balls in the urn before and after each iteration -- the loop invariant, if you wish. After termination of the process, the invariant together with the fact that one ball is in the urn should tell us the color of that ball. Since there is finally one ball, and one is an odd number, we are led to consider whether the parity of the number of black (say) balls remains invariant; after all, parity is a simple property, and simplicity is what we should always strive for. A look at the process indicates that the parity of the black balls is not an invariant. But a look also convinces us that the parity of the white balls is invariant: the number of white balls is reduced by 2 or not at all. Hence the final ball is white if and only if initially there is an odd number of white balls.

This little problem was solved by myself and others in under 10 minutes. I have, however, watched others struggle for over a half hour or more, until a hint was given to look for an invariant, at which point the answer came in 5 more minutes. Clearly, the idea of invariants, of looking for properties that remain true throughout a process, is extremely important to have in our bag of tools! But it is only effective if one has practiced using it and consciously attempts to apply it.

It is often claimed that an invariant is too difficult to derive, especially before having written a loop. At the beginning this is certainly true, but as one gains experience it becomes easier and easier and, finally, becomes a habit. One can introduce the concept to new

programmers very early, but without any formalism, as follows. I tell my students to group variable declarations by logical relationship (instead of by type). Secondly, I tell them to define their variables _before_ writing any statements that use them, where by "define" I mean "describe their logical relationship". For example, if an array T is to contain a list of numbers representing temperatures, and K is a "counter" (such vague words should be struck from our vocabulary, for they only encourage imprecision) for the list, then they should _first_ write down the sentence "T[1:K] is the list of temperatures created thus far." Once this has been written down, one sees immediately that the initialization K:=0 is called for, and whenever one has to write a program statement dealing with T or K one can refer to this definition. The theory has really helped me here, for I view this definition as nothing more than an invariant that must hold at (almost) all places in the program. This shows clearly how the theory can lend insight and can be put into practice in an informal manner.

5. SEARCHING FOR SIMPLICITY

Mathematicians and computer scientists often have the problem of presenting their work in such a way that others can understand, but with enough formality so that one can see that the work is complete and correct. The mathematician has to find the right balance between formality and detail on the one hand and intuition on the other; he has to find the _simplest_ way to express exactly what is necessary to convey understanding. The programmer has an even harder problem in this regard, because of the overwhelming mass of detail that programs entail. Hence it is even more important for the programmer to search for just the right kind and amount of detail -- no more, no less -- so that a reader (and the programmer himself) can understand and appreciate as quickly and easily as possible. The programmer tries to find notation and organization in order to make complexity tractable -- or even to make sure complexity never arises.

This is so important that I like to call the field of programming _computational simplicity_, as opposed to the already-existing field in computer science, computational complexity.

This need for a balance and simplicity is rarely taught, and some would argue that it can't be; a person either learns it through experience and interaction with others or doesn't. Nevertheless, it should be stressed again and again to the student and illustrated by example. The student can benefit from detailed feedback on programs, in the way of comments explaining how a slightly different technique, or a more precise comment, or a more suitable notation would have helped. This takes time and effort on the part of the teacher; it cannot be done with an automatic program grader (these things I would throw out). But it is worth it. The problem is, of course, that not all programming teachers understand this need for continually searching for simplicity.

5.1 _An example_: _exponentiation_

Consider the well-worn problem of calculating b^c for $b, c \geq 0$ (with $0^0 = 1$). A programmer who understands exponentiation and binary arithmetic might note that

$$b^{13} = b^8 b^4 b^1,$$

and that $13 = (1101)_2$. He could then arrive at a formula for b^c based on the binary representation for c:

$$c = (c_{n-1} \cdots c_1 c_0)_2$$

$$b^c = \prod_{i=0}^{n-1} (\text{if } c_i = 1 \text{ then } b^{2**i} \text{ else } 1)$$

where 2**i means 2^i and each c_i is 0 or 1. It seems plausible, therefore, to think of using a loop with a "counter" k running from 0 to n; initially $z = 1$, and at each iteration the kth factor (if c_k then b^{2**k} else 1) is "multiplied into" z.

For reasons of efficiency, a variable x is introduced to contain the value b^{2**k} and a variable y is introduced to contain that part of the binary representation of y still to be "processed". This leads to the following loop invariant P:

$$P:\ 0 \leq k \leq n,$$
$$z = \prod_{i=0}^{k-1} (\text{if } c_i = 1 \text{ then } b^{2**i} \text{ else } 1)$$
$$y = (c_{n-1} \cdots c_k)_2$$
$$x = b^{2**k}$$

Now one can develop the following loop:

```
x, y, z, k:= b, c, 1, 0;
do k ≠ n →
    if even(y) → skip
     □ odd(y)  → z:= z*x
    fi;
    k, x, y:= k+1, x*x, floor(y/2)
od
```

But let us step back for a minute and see whether everything is as clear as it could be. Note that we are forced to refer to the number of bits of c. Moreover, the invariant includes a complicated product of terms, which might be a major stumbling block. Is it necessary? Can we simplify?

In a sense, z contains part of the final answer: $z*Z = b^c$ for some Z, and Z is the product of the terms (if $c_i = 1$ then b^{2**i} else 1), for i between k and n-1. Some simple formula manipulation leads us to the neat fact that

$$Z = x^y$$

Hence, $z*x^y = b^c$, and the invariant P can be written -- without referring to n or k -- as

$$P1: y \geq 0 \wedge z*x^y = b^c.$$

Redevelopment of the algorithm using P1 as the invariant leads to

```
z, x, y:= 1, b, c;
do y ≠ 0 → if even(y) → y, x:= y/2, x*x
           □ odd(y)  → y, z:= y-1, z*x
           fi
od
```

This is the well-known logarithmic (in c) algorithm that is usually presented. Rearrangement can make it slightly more efficient by reducing unnecessary tests. What I have attempted to show is that a typical programmmer could have developed it as a matter of course, _if_ he understood the use of invariants and formal correctness ideas, and _if_ he relentlessly pursued simplicity and a good balance between formalism and common sense. To understand the final algorithm, a reader must also understand correctness ideas.

Note that the algorithm has not been proved formally correct in all details; to do this would just lead to obscurity. Rather, only that part of the formalism necessary to promote understanding -- especially the loop invariant -- has been given, while other parts deemed to be obvious enough or easy enough have been left to the reader.

This search for simplicity, the right amount of detail, for the balance between formalism and intuition, must be relentlessly pursued. At each step, the programmer must ask himself: have I found the right way to express things? Along with this goes the need for style and elegance -- something else that is rarely even talked about. Elegance _does_ matter; only if we continually try to find the neatest, simplest argument or notation can we hope to begin to program well. Examples of elegant programs and arguments must be continually pointed out to the student.

6. USING SUITABLE NOTATIONS

In my introduction, I made a rather caustic remark about BASIC and FORTRAN. Let me temper that remark somewhat with an explanation. We have been struggling for 30 years to find the right notations to express algorithms. This struggle has been intensified by the fact that we want our programs to be executed on computers, which means the notation must be implemented. Much effort goes into implementing a notation, and, when a new, perhaps better, notation arises, we resist, simply because of inertia. Not only is it difficult to change our thinking habits, but all our old programs have to be changed too. I can therefore (almost) understand the resistance to change.

Throughout history, there has been a continual battle over notation, and the present is no different in this regard than the past. For example, in the 1600s and 1700s the following signs were used for equality:

= ꭗ || | 2|2)=([.

And in the early 18th century there was a long, drawn-out battle between the semi-finalists, Robert Recorde's = and René Descartes's ꭗ, with = finally emerging victorious. From this standpoint, it is too bad that the universally-adopted sign = is used for other purposes in some languages, viz. assignment. (In his book on the history of notation [6], Cajori mentions that ꭗ is probably the symbol for Taurus on its side. Secondly, after hearing me talk on this subject, Wlad Turski conjectured that = came from the astronomical sign ♎ for the equinox (when night and day are of _equal_ length). Since at one time the sun was in Taurus during the vernal equinox, both = and ꭗ could have the same origin!)

One also remembers the struggle in the 1600s between two notations for the derivative:

Newton's $\dot{y}$ and Leibnitz's dy/dx, with Leibnitz'z superior notation winning.

Finally, mathematics had its Ken Iverson in those days, in the person of William Oughtred, who used over 150 mathematical symbols, while at the same time Robert Simon still clung to using only natural language for proofs and expositions, using not *one* mathematical symbol in his 1756 edition of Euclid.

In his book, Cajori concludes that the whole fight over symbolism or notation is a good object-lesson to mathematicians, and that nothing is all good or bad. Used in moderation and at the right time, symbolism is at its best.

This is precisely what we were talking about in the previous section: a proper balance is needed.

Hence, I don't get too upset when people use FORTRAN or BASIC; I just feel sorry for them.

But what does upset me is to hear of programmers that know *only* FORTRAN or BASIC. Language shapes the thought and mind, said Benjamin Whorf, and programming languages (notations) do the same. One who knows only FORTRAN or BASIC is severely hindered in his thinking about programming, because he must place all his algorithmic thoughts in one notation, and in an inadequate one at that.

It has been shown that some programs simply cannot be written with the FORTRAN do-loop, and hence people who know only that one notation for looping are severely handicapped. One person recently told me that he likened a person who has been forced into knowing *only* FORTRAN to a child having been shut up in a dark closet for most of his life.

It is also obvious that some notations are clumsier than others, some express our intentions better than others, and some are more error-prone than others.

It therefore behooves the professional to learn several notations, and to choose which one(s) he wants to work with based on *technical* considerations, and not just on familiarity. A programmer may be forced to use a certain notation at times, but that does not mean he must *think* in it. As I said many years ago, one programs *into* a language, not *in* it.

7. SUMMARY

This completes the major part of my talk; I would now like to summarize. I have discussed two major technical areas in which a programmer should be competent: logic and a theory of programming. And I have discussed some more elusive ideas that should take root: the need for simplicity, the need for elegance, the need for balance between formalism and intuition, the need for a continual search for the right notation. While I cannot guarantee it, my opinion is that a programmer who takes my advice seriously will increase his programming effectivity.

But I add a caveat to my remarks; learning programming is hard work. No one-day course on the subject can hope to accomplish much. It takes study and practice. Our old habits, developed through years of purely operational thinking due to our computers, have to some extent to be overcome. Moreover, it is not enough just to know the technical material, one must consciously attempt to apply it. Much of the material seems quite simple after a while, but it cannot be applied until one *tries* to apply it. In this regard, I like the following saying -- I don't know to whom I should attribute it:

> Never dismiss as obvious any fundamental principle, for it is only through *conscious* application of such principles that success will be achieved.

But, in the long run, I think the work one must put in to learn such material is eminently worth it.

ACKNOWLEDGEMENT

I wish to thank Fred Schneider, of Cornell, and Peter Deussen, of Karlsruhe, for many excellent, constructive criticisms of drafts of this manuscript.

REFERENCES

[1] Buxton, J.N. et al (eds.) Software Engineering. Petrocelli,]975. (Report on two NATO Conf.)
[2] Gries, D. An illustration of current ideas on the derivation of correct programs. IEEE Trans. Soft. Eng. 2 (Dec 1976),238-244.
[3] DeMillo, R.A., R.J. Lipton, A.J. Perlis. Social processes and proofs of theorems and progrms. CACM 22 (Mar 1979), 271-280.
[4] Austing, R.H. et al. Curriculum '78. CACM 22 (Mar 1979), 147-166.
[5] Dijkstra, E.W. A Discipline of programming. Prentice Hall, 1976.
[6] Cajori, F. A History of Mathematical Notation. Open Court. Publ. Co. LaSalle, 1974.