

WHAT A PROGRAMMER SHOULD KNOW ABOUT LOGIC

D. Gries

Rapporteur: Dr. C.M. Holt
Mrs. M. Koutny

What Programmers Don't and Should Know

David Gries

Computer Science Department, Cornell University

prepared for the

Twentieth University of Newcastle upon Tyne International Seminar
on the Teaching of Computing Science at University Level

titled

Logic and its Application to Computing Science

Table of Contents

0. Introduction, 1
1. Logic, 2
2. Writing formulas in the logical notation, 4
3. The syntactic proof, 6
4. Developing correct programs, 9
5. The hazards of examples, 10
6. Further comments on notation, 12
7. Conclusions, 13
8. Acknowledgements, 13
9. References, 13

Introduction

I am concerned with the way algorithms and programs are developed and presented in the research papers that we write, in the texts from which we teach, and in 'everyday' programming. Generally speaking, the level of professionalism is far below what it could be, given the advances in the state of the art and science of programming over the past fifteen years,

It is doubtful that older computer scientists will change their methods of developing and presenting algorithms. The development and presentation of an algorithm is often the secondary concern to them; the primary concern is the area in which the algorithm is being written, and that is where most of the learning and research efforts are placed. (Also, it is difficult, for all of us, to change our

ways —I certainly don't want to. As Mark Twain said, 'Nothing so needs reforming as other people's habits'.)

Thus, it is to the younger generations that we turn for improvement. If we teach them more effective ways of thinking, of developing and presenting algorithms, they in turn will press for and effect the necessary changes.

But what should we give today's students that they are not already getting? First, we should be instilling in them

- An appreciation of the need for precision, rigor, and elegance when dealing with specifications of an algorithm, when analyzing the properties of the objects being manipulated by the algorithm, when developing it, and when presenting it.

Of course, an appreciation of a need is not enough; one must be able to *fill* the need. Basic to this is

- An in-depth experience with the propositional and predicate calculi, the aim being a proficiency and agility in formal manipulation according to their axioms, inference rules, and theorems.

The emphasis here is not on deep theorems about consistency, completeness, non-standard models, and the like. Instead, it is on the use of logic as a tool in our everyday striving for simplicity and elegance. The *syntactic proof* as a sequence of formal manipulations according to a logic should assume importance. Through many and

varied examples, it should become clear that formal manipulation is a useful—even indispensable—*tool* of the programmer. Out of this will come

- An appreciation of the importance of suitable, simple notation that is geared to formal manipulation and a lessening of the need for examples as a means of conveying understanding.

One may argue that these things are already being taught and practiced. Perhaps, but far less than they should be. In the U.S., they are to be conveyed in a discrete mathematics course, where the student is first introduced to the predicate calculus, but in texts for these courses the emphasis is more on facts and knowledge than on method and appreciation. Rarely is the predicate calculus used outside the single chapter devoted to it, rarely is there a discussion and comparison of notations, and rarely are different proofs or proof methods compared.

Further, relatively few articles and texts on data structures and algorithms make use of what we know about programming and the description of algorithms, so any appreciation absorbed earlier is not reinforced.

In summary, more emphasis should be placed on mathematical tools and methods, and in such a way that an appreciation for their need and for elegance and simplicity, as well as a sense of discrimination, is inculcated in the student. Below, I want to explain in more detail what I mean, using examples from both mathematics and programming and giving reasons why one method or notation might be preferred over others. Formal methods for program correctness are of course discussed, but the student needs to master some basic mathematics before these can be appreciated and applied.

We begin with a brief overview of the propositional and predicate calculi.

1. Logic

Two reasons for using a formal system of logic in our work are:

- To be able to state concepts (for example, mathematical induction) or statements (for example, a specification of a program) clearly and unambiguously.
- To be able to give shorter, simpler, and more elegant proofs and derivations and to increase our powers of reasoning; the proofs are for human, rather than machine, consumption.

Let us discuss logic briefly in light of these reasons.

The propositional calculus

Basic to the propositional calculus is a set of axioms and inference rules that allow the manipulation of formulas. A *theorem* is an axiom or a formula that can be generated from the axioms using the inference rules. I see two general approaches to such proofs, equational reasoning and natural deduction.

In the equational-reasoning approach, the main inference rules include substitution of equals for equals, transitivity of equality, and modus ponens. These rules allow us to use axioms and theorems like De Morgan's Law ($\neg(X \vee Y) \equiv \neg X \wedge \neg Y$) and Associativity of Disjunction to translate one formula into an equivalent one (or into one that is implied by the first). For example, the following proof shows, by the law of transitivity of equality, that $b \supset c \equiv \neg c \supset \neg b$:

$$\begin{aligned} b \supset c & \\ \equiv \neg b \vee c & \quad (\text{Implication}) \\ \equiv c \vee \neg b & \quad (\text{Commutativity}) \\ \equiv \neg\neg c \vee \neg b & \quad (\text{Double negation}) \\ \equiv \neg c \supset \neg b & \quad (\text{Implication}) \end{aligned}$$

Each line of the proof follows from the previous one by a substitution of equals for equals using the axiom or theorem of equality given to the right. One often uses more than one rule in going from one line to the next, attempting to achieve brevity without sacrificing understanding.

Remark. Logicians tend to use the symbol \supset for implication. Using X and Y to denote the sets of states in which x and y are true, we have $x \supset y$ iff $X \subset Y$, which is indeed a confusing use of symbols. Thus, we prefer a different symbol for implication. \square

The proof can also be written as follows, where each new formula may be preceded by a line that describes the reason the transformation is valid; this is useful when the reason may be more complicated and may take more space.

$$\begin{aligned} b \supset c & \\ \equiv \text{(Use the Law of Implication)} & \\ \neg b \vee c & \\ \equiv c \vee \neg b & \\ \equiv \text{(Double negation)} & \\ \neg\neg c \vee \neg b & \\ \equiv \neg c \supset \neg b & \end{aligned}$$

In a *natural deduction* system, introduced by Gerhard Gentzen in the 1930's, no formulas are assumed to be axiomatically valid; there are only rules of inference. To compensate for the lack of axioms, it is permitted to introduce any formula as a hypothesis at any stage. Generally, such a system has two kinds of inference rules for each operator, introduction rules and elimination rules. One kind introduces the operator, the other eliminates it. For example, the rules for \wedge are

$$\wedge\text{-I: } \frac{X, Y}{X \wedge Y} \quad \text{and} \quad \wedge\text{-E: } \frac{X \wedge Y}{X}, \frac{X \wedge Y}{Y}$$

The first rule indicates that, for any formulas X and Y , from the assumptions X and Y the formula $X \wedge Y$ can be inferred. The other two indicate that from $X \wedge Y$ both X and Y can be inferred.

As in the equational-theory approach, a proof consists of a sequence of lines; each is an assumption, an axiom, or an instance of the result of an inference rule for which the corresponding instances of the hypotheses appear on previous lines of the proof. To the right is stated the inference rule being used and the numbers of previous lines containing its assumptions. Here is a proof:

From $p \wedge q$ infer $q \wedge p$		
1	$p \wedge q$	assumption 1
2	p	$\wedge\text{-E, 1}$
3	q	$\wedge\text{-E, 1}$
4	$q \wedge p$	$\wedge\text{-I, 3, 2}$

The natural deduction system is thus called because it is supposed to mimic the way we 'naturally' reason (perhaps that is a good reason for eschewing it!), implying that we naturally think of introducing and eliminating operators. Nevertheless, few mathematicians and computing scientists use such inference rules as a formal tool in their work, for they are just too cumbersome. However, the approach has been extremely useful for *studying* logic and is becoming more and more useful in mechanical theorem proving and, for this reason, the computing scientist should be familiar with it. And some systems for doing formal mathematics on the computer are indeed becoming very useful tools and could be helpful in teaching students about formalism and its uses; I refer for example to Constable's system *PRL* [9].

One can introduce the substitution-of-equals-for-equals rule as a meta-rule of a natural-deduction system, thus merging the equational and natural-deduction approaches. Therefore, we need not worry about which is more powerful, etc. Instead, we should be looking at how informal proofs can be written using a mixture of the two methods to arrive at the best proofs. In general, the equational approach does tend to lead to shorter, more readable, proofs.

The above proofs are *syntactic* in nature, because they are simply a syntactic manipulation of formulas without regard to their meaning. My opinion is that we should be striving more and more for such syntactic proofs, and I will have more to say on this later.

A few colleagues have mentioned that *formal* might be more appropriate than *syntactic*, in the sense I am using it. I prefer *syntactic* because it emphasizes more the

complete abstraction of the meaning of the symbols that are being manipulated in a proof, an important property that has to be made clear.

Our use of the propositional calculus fits in nicely with our notion of computers and states. During execution of a program, the computer is in a *state*, which contains a value for each variable. The state is therefore a function from variables to values. Using s to denote the state, if variable x has value v in state s , then $s.x = v$. Thus, the notion of a *model* for the calculus arises naturally. Unfortunately, programmers are usually taught *only* the model, in that they are taught how to evaluate Boolean expressions but not the rules for manipulating them.

Remark. I use '.' to denote function application, as an experiment to see whether its use reduces the number of parentheses in formulas, thus making manipulation easier. Function application binds tightest, so $f.x+2 = (f.x)+2$. We write the application of a function of two arguments as $g.(x, y)$ or, using currying, as $g.x.y$. \square

The predicate calculus

The introduction of predicates allows variables of other types to be used, and with this we introduce quantified expressions. My discussion here deals mainly with the notation used for quantified expressions and the calculus of axioms and inference rules to be used.

Typically, mathematicians use a notation like $\exists x. P$, or perhaps $\exists x \in X. P$, to stand for 'there exists a value x such that P holds'. Following (but deviating slightly from) Dijkstra, I prefer instead the notation

$$(0) \quad \exists(x: R: P)$$

where R is a predicate specifying the range of values x under consideration. Actually, (0) can be written in the more conventional notation as $\exists x. R \wedge P$, so I need to substantiate my use of it. There are several reasons for it.

First and foremost, our notation should be geared to our manipulative needs. Often, in programming and related fields, we manipulate quantified expressions in which the range R remains constant —e.g. it is the subscript range of an array— while P changes. Making the range distinct allows us to show this more clearly. It allows us to introduce conventions to eliminate the range, thus reducing what has to be written. For example, an omitted range is assumed to be the same as on the previous line, as in

$$\begin{aligned} & \exists(x: R: P) \\ \equiv & \exists(x: P) \end{aligned}$$

Also, we often find ourselves *splitting* a range:

$$\begin{aligned} \forall(i: 0 \leq i < n+1: P) & \equiv \\ \forall(i: 0 \leq i < n: P) \wedge P_n^x & , \end{aligned}$$

and using form (0) instead of the form $\exists x.R \wedge P$ allows us to develop inference rules for manipulating the range. Thus, we are choosing notations geared to our manipulative needs and suggestive of our problematic concerns.

Remark on notation for textual substitution. I am using R_x^e to denote a copy of R with all free occurrences of x replaced by e . Many other notations are used for textual substitution, including R_x^e , $R(x/e)$ and $R(e/x)$. For a linear notation $R[x:=e]$ is a suitable choice because of the connection between assignment and textual substitution; the assignment statement axiom would read

$$\{R[x:=e]\} x := e \{R\}$$

Next, we require the parentheses in the new notation because of the importance of the scope-introducing concept. The beginning of the scope of the newly introduced variable is usually clear; the end of the scope should be just as clear.

Finally, let me discuss the use of the quantifier outside the parentheses. We can view the notation

$$x: R: P$$

as simply a *scope-introducing* mechanism, in which x is the variable introduced, R is a predicate giving its range, and P is some expression or statement, possibly containing free occurrences of x . We can apply operators to the scope:

0. $\exists(x: R: P)$
1. $\forall(x: R: P)$
2. $N(x: R: P)$
3. $\Sigma(x: R: E)$
4. $\Pi(x: R: E)$
5. $MAX(x: R: E)$
6. $\lambda(x: R: E)$
7. $BLOCK(x: R: S)$
8. $for(x: R: S)$
9. $\{x: R: E\}$

In cases 0-2, P is of type Boolean. Case 2 denotes the number of values x in the range R such that P holds. In cases 3-5, E is integer or real-valued; these produce the sum, the product, and the maximum over values E such that R holds. Case 6 is a function that yields the value E_a^x given an argument a . Case 7 is an Algol-like block that introduces a new variable; case 8 is a loop, executing statement S for all values of x in the range R . Case 9 is the set of values E where x ranges over values for which R is true.

It is the separation of the range R from the expression P and the explicit introduction of the fresh variable, the dummy, that allows us to unify in this fashion.

Operator N (case 2) can be used to simplify what might otherwise be very awkward. For example, the following statement says that sequence b is a permutation of

sequence c :

$$\forall(v:: N(i: 0 \leq i < \#b: b.i = v) = N(i: 0 \leq i < \#c: c.i = v))$$

i.e. each value occurs the same number of times in b and c . Try to express this statement without using operator N .

Suitable axioms for quantification

The student must learn the rules for manipulating quantified expressions. These seem difficult at first, but practice quickly leads to internalization. Useful axioms and theorems of equivalence, geared to our manipulative needs, appear in [2] and [3]. We list some for the quantifier \forall . We will be using such axioms subsequently.

0. De Morgan: $\neg \forall(x: P: Q) \equiv \exists(x: P: \neg Q)$
1. $\forall(x: false: P)$
2. $\forall(x: P \wedge Q: R) \equiv \forall(x: P: \neg Q \vee R)$
3. $\forall(x: P: R) \Rightarrow \forall(x: P: Q \vee R)$
4. Dummy renaming:
 $\forall(x: P: Q) \equiv \forall(y: P_y^x: R_y^x)$
(where y is a fresh variable)
5. One-point rule: $\forall(x: x = E: P) \equiv P_E^x$
(where x is not free in E).
6. Range-splitting: $\forall(x: P \vee Q: R) \equiv \forall(x: P: R) \wedge \forall(x: Q: R)$

2. Writing formulas in the logical notation

The student must begin to feel that logic is useful: he needs *motivation*. The presentation of many examples of the formalization of statements and concepts will help. We give some examples here.

Induction and well-foundedness

First, let us consider the concept of complete mathematical induction. Typically, this is stated as follows: Let $P.x$ be a predicate with argument x , where x ranges over the natural numbers. Suppose we prove the base case: $P.0$ holds. Suppose we prove the inductive case: for all $y > 0$, if $P.x$ holds for all x less than y then $P.y$ holds. Then we conclude that $P.x$ holds for all natural numbers.

We now generalize and formalize this statement. First, let U be a set of values and let $<$ be any binary relation over U . We say that $(U, <)$ admits induction if the following holds. Let $P.x$ be a predicate, with x ranging over U . Then $P.x$ holds for all $x \in U$ iff, for any y , the truth of $P.x$ for all x less than y implies the truth of $P.y$. We formalize this as follows:

- (1) **Complete mathematical induction:**
 $\forall(x: x \in U: P.x) \equiv$
 $\forall(y: y \in U: \forall(x: x < y: P.x) \Rightarrow P.y)$

The equivalent formula (2) does not use implication. To show that (1) \equiv (2), use the Laws of Implication, De Morgan, and Commutativity of \vee . In (2) and subsequent discussions of induction, the range of a quantified expression is omitted if it is the universe U .

- (2) **Complete mathematical induction:**
 $\forall(x:: P.x) \equiv$
 $\forall(y:: P.y \vee \exists(x: x < y: \neg P.x))$

Note that these formulas are equivalences and not implications, although the original informal statement of induction over the natural numbers was couched as an implication. The stronger and commutative equivalence is preferred over the weaker and non-commutative implication.

Note also that (1) and (2) do not distinguish between the base case and the inductive case. In general, avoid case analysis like the plague; even reducing two cases to one is a worthwhile simplification.

It has been claimed that induction should be described in terms of a base case and an induction case because that's how we use it. Before agreeing with this claim, decide whether our traditional two-case view has been forced on us by the two-case formulation. Perhaps our formal proofs using induction won't need two cases!

Let us express the related concept of a well-founded set in our notation. Let S be a subset of U . An element y in S is called a *minimal element* of S if no element smaller than y (with respect to $<$) is in S ; i.e. if

$$y \in S \wedge \forall(x: x < y: x \notin S)$$

$(U, <)$ is well-founded if every nonempty subset of U contains a minimum element, if for every subset S of U the following holds:

- (3) **Well-foundedness:** $\neg \text{empty}(S) \equiv$
 $\exists(y:: y \in S \wedge \forall(x: x < y: x \notin S))$

Later, we shall return to the notions of induction and well-foundedness and prove them equivalent.

Specifications in programming

One reason for using a formal notation is to be able to make precise and clear what is ambiguous or confusing. Consider the following statement: every value of $b.(i..j)$ that is not in $b.(h..k)$ is in $b.(h..k)$. (Here, b is an array and $b.(i..j)$ denotes the segment of b consisting of $b.i$ through $b.j$.) Such contorted and confusing statements do appear in informal specifications of programs. We can place this in our notation as follows:

- (4) $\forall(v: v \in b.(i..j): v \notin b.(h..k) \Rightarrow v \in b.(h..k))$

Of course, it may still seem confusing, but let us now simplify it:

- (4)
 \equiv {Law of Implication}
 $\forall(v:: \neg v \notin b.(h..k) \vee v \in b.(h..k))$
 \equiv $\forall(v:: v \in b.(h..k) \vee v \in b.(h..k))$
 \equiv $\forall(v:: v \in b.(h..k))$

Thus, the original statement is equivalent to the much simpler 'every value in $b.(i..j)$ is also in $b.(h..k)$.' For the record, typically two thirds of the students shown this problem are not able to deal correctly with (4) *unless and until* they apply the formal rules of manipulation. Typically, the student says that (4) is equivalent to *false*.

Let me now specify an algorithm. The form and content of a specification of an algorithm colors —indeed, provides the insight for— algorithmic development. So it makes sense to specify the algorithm precisely and simply before beginning the development.

Consider an integer array $b.(0..n-1)$, where $n \geq 0$. Consider any segment $b.(i..j-1)$ of b ; we can compute the sum of its elements. We want an algorithm that finds the maximum such sum over all segments of b .

First define the sum of a segment:

$$S_{i,j} = \Sigma(k: i \leq k < j: b.k)$$

Next, give the result assertion R of the algorithm:

- (5) $R: m = \text{MAX}(i,j: 0 \leq i \leq j \leq n: S_{i,j})$

Difficult this was not. Later, we will see how specification (5) will guide the algorithmic development. But for now, we can discuss one advantage of having written (5).

What is the maximum sum over all segments of the array $b = (-1, -8, -4)$? Ask your students; if they depend on the informal specification, half will say -1 and the other half 0 , depending on whether they think of the empty segments as belonging to b . The formal specification makes clear what is meant: empty segments are included, so that the answer is 0 .

Note that $S_{i,j}$ does not include the value $b.j$. For our purposes, for our formal manipulations, ranges like $i \leq k < j$ are often preferred over ranges like $i \leq k \leq j$. Arguments concerning this have been put forth by Dijkstra [4], [5]. Here is his argument for inclusion of the lower bound and exclusion of the upper bound:

Exclusion of the lower bound, as well as inclusion of the upper bound, would have forced bounds outside the realm of the natural numbers. I intend to let a lower bound never exceed an upper bound; then we have the advantage that two ranges can be joined to form a single one if the upper bound of the one equals the lower bound of the other. Finally, the number of elements in the range equals the difference of the bounds. It now stands to reason to identify the elements of a sequence of length M —rows of a matrix or characters of a string— by a subscript in the range

$0 \leq \text{subscript} < M.$

Let me give one more example of a specification of a program, one that leads directly to the algorithm. Given is a nonempty sequence $b.(0..n-1)$ of integers that, lexicographically speaking, is not the largest (e.g. (8, 5, 3, 3) is the largest sequence that can be built with the bag (3, 3, 5, 8)). Write an algorithm that changes the sequence into the next largest one using the same integers. This seems like a difficult algorithm until we specify more precisely what the next largest sequence is. First, let j be the index of the leftmost element of b to change. Since this element is to become larger and is replaced by something to its right, j is the largest value satisfying

$b.j < b.(j+1) \wedge$
 $b.(j+1..n-1)$ is a non-increasing sequence.

or $j = \text{MAX}(k: k < n-1: b.k < b.(k+1))$

Next, one must specify which value $b.i$ of $b.(j+1..n-1)$ is to be placed in $b.j$. Since the next largest permutation of b is to be created, $b.i$ should be the smallest value of $b.(j+1..n-1)$ that is greater than $b.j$. Since $b.(j+1..n-1)$ is non-increasing,

$i = \text{MAX}(k: k < n: b.j < b.k).$

By the definition of i , and since $b.(j+1..n-1)$ is non-increasing, the sequence $b.(j+1..i-1) \wedge b.j \wedge b.(i+1..n-1)$ is also non-increasing. Since the result desired is the smallest permutation of b that is larger than the initial b , we can specify the algorithm as implementing the assignment

$b := b.(0..j-1) \wedge b.i \wedge$
 $\text{reverse}(b.(j+1..i-1) \wedge b.j \wedge b.(i+1..n-1))$

where j and i are defined above.

In this instance, refining or detailing the specification has led to a specification in which the algorithm to use is more apparent.

This ends our short excursion into the propositional and predicate calculi and notations to be used in formulas. We have attempted to show that formalizing informal statements can lead to simpler, clearer, and more precise statements, thus increasing our understanding. Further, the formal notations we use will be a factor in how simple our formalizations will be, so they must be chosen carefully. They should be geared to our manipulative needs, which means that they and the manipulation rules we use should be designed to minimize the text we must write.

The manipulation of formulas, of course, is aimed at proving properties of the objects being dealt with. Let us now turn to a discussion of such proofs.

3. The syntactic proof

We mentioned earlier the notion of a syntactic proof: a sequence of symbolic transformations according to given rules. The notion of such a proof has been around for some time, having been championed by David Hilbert, one of the greatest mathematicians of all time. In fact, Hilbert's program was to formalize *all* of mathematics as a set of axioms (or axiom schema) and inference rules.

Students may be taught about an axiom-inference-rule system, but rarely are they shown its real power; rarely is it used for anything interesting. Students are simply not taught to consider it a useful tool. More and more, I view the syntactic proof as a necessity, as a way of forcing myself to achieve a rigor and simplicity and understanding that I would not otherwise achieve. Especially in programming, with the myriad of details that have to be manipulated and understood, it is essential to attempt to strive for the rigor and precision that the syntactic proof requires.

Some question the use of the syntactic proof, feeling that it is cumbersome, complex, and difficult exactly in programming, where there are so many details. It requires symbol manipulation, and that is a task best left to the computer, for humans are bad at it. I disagree with this viewpoint, and I generally find that the work needed to produce a syntactic proof can lead to simpler proofs and better understanding.

Let me quote Hilbert, from his famous lecture at the Second International Congress of Mathematicians in Paris in 1900, in which he outlined his famous 10 problems (23 in the manuscript, but only 10 mentioned in the lecture):

It remains to discuss briefly what general requirements may be justly laid down for a solution of a mathematical problem. I should say first of all, this: that it be possible to establish the correctness of the solution by means of a finite number of steps based upon a finite number of hypotheses that are implied in the statement of the problem and that must be exactly formulated. This requirement of logical deduction by means of a finite number of processes is simply the requirement of rigor in reasoning. Indeed, the requirement of rigor, which has become a byword in mathematics, corresponds to a universal philosophical necessity of our understanding ... only by satisfying this requirement do the thought content and the suggestiveness of the problem attain their full value. ...

It is an error to believe that rigor in the proof is the enemy of simplicity. On the contrary, we find it confirmed in numerous examples that the rigorous method is at the same time the simpler and the more easily comprehended. The very effort for rigor forces us to discover simpler methods of proof. It also frequently leads the way to methods that are more capable of

development than the old methods of less rigor. ...

Wherever mathematical ideas come up, ..., the problem arises for mathematicians to investigate the principles underlying these ideas and so to establish them upon a simple and complete system of axioms ... (Quoted from *Hilbert*, by Constance Reid, Springer Verlag, New York, 1983.)

During his time, Hilbert's program for formalizing mathematics received its share of criticism, with some mathematicians objecting to his 'reducing the science to a meaningless game played with meaningless marks on paper'. However, it is precisely the shuffling of meaningless symbols according to given rules that provides confidence! Couched in terms of English and informal mathematics, an argument may be difficult to understand and ambiguous. Once we agree on the formulation of the problem in a formal notation, then checking a well-written formal proof requires only checking that each rule was correctly applied.

Criticisms are similarly made about the use of formalism in programming. In the past, these criticisms were partially valid, in that we had difficulty ourselves in using the formalism ourselves. However, advances are such that it is clear from the literature that formalism can be used to great advantage in developing algorithms. At this point, it is only lack of education that is hindering progress.

Students—and anyone not experienced with symbol manipulation—have great difficulty at first in applying formal methods. Once they have made a textual substitution, say, they no longer 'understand what the formula means' and are hesitant to make further manipulations for fear of making mistakes. And because of this, they equate rigor with rigor mortis, with a stiffening of their abilities.

I have given courses in which I had actually to guide a student's hand as the student made its first textual substitution in connection with using the assignment statement axiom! This was depressing, for symbol manipulation is what programming is all about. Such courses, expected to be on the development of programs, had to spend far too much time on the predicate calculus and symbol manipulation. And these were not dumb or inexperienced students; they simply hadn't received a proper education.

The cure for this is a study of the use of logic as a tool in our work, with many examples chosen to illustrate the effectiveness of the approach; enthusiasm on the part of the instructor; a concentration on improving penmanship, for it is very important in reducing careless mistakes; and lots of practice.

Let me now turn to an extended, important, example of the syntactic proof.

Induction and well-foundedness again

We now turn to the proof of equivalence of mathematical induction over a partially ordered set $(U, <)$ and the well-foundedness of $(U, <)$. (I am indebted to Edsger W. Dijkstra for the proof.) We have already given formal definitions of mathematical induction (2) and well-foundedness (3). To prove their equivalence, we need only show how to transform one into the other using substitution of equals for equals. This is done in Fig. 0 on the next page—what a simple proof!

The result itself is particularly noteworthy because it shows that mathematical induction has a formal basis; induction can be applied whenever (and only then) the universe upon which it is to be applied is well-founded. The result is completely general; one doesn't need a different argument to allow induction on natural numbers, trees, lexically ordered pairs, lengths of derivations in a grammar, etc; one need only know that the set under consideration is well-founded.

Finally the result shows the student that mathematical induction has a firm basis; we know precisely the conditions under which it can be used.

In light of all these advantages, it is disheartening to note that few of the current texts in discrete mathematics contain any proof about when induction can be used, much less this one. Most of the texts simply state the principle of mathematical induction as a *Grand Principle* that the student should believe and absorb without questioning its validity. And arguments concerning induction over other sets are left to the imagination of the reader.

Interestingly enough, some computer scientists dislike this syntactic proof, feeling that it does not convey the 'intuition' behind induction. (Some also do not like formulation (2) because it does not separate the base case from the other cases, or do not like writing (2) as an equality instead of as an implication; that is a different story.)

I asked one such colleague to give me his proof of equivalence; it is shown in Fig. 1—I have changed notation and rewritten a bit, but not much. The proof relies on the finite-chain property, which we now explain. A *decreasing chain* is a sequence x_0, x_1, x_2, \dots of elements of U such that $\forall(i: 0 \leq i: x_{i+1} < x_i)$. The finite-chain property states that all decreasing chains have finite length:

(6) **Finite-chain property:** $\forall(y :: DCF.y)$

where function *DCF* is defined as

$DCF.y =$ 'Every decreasing chain beginning with y is finite'

$\neg \text{empty}(S) \equiv \exists(y :: y \in S \wedge \forall(x : x < y : x \notin S))$ (This is (3))
 ■ (Complement both sides, use Law of Negation and DeMorgan's Law)
 $\text{empty}(S) = \forall(y :: y \notin S \vee \exists(x : x < y : x \in S))$
 ■ (Define a predicate $P: P.x = (x \notin S)$ and replace occurrences of S by P)
 $\forall(x :: P.x) = \forall(y :: P.y \vee \exists(x : x < y : \neg P.x))$ (This is (2))

Figure 0. Syntactic proof of induction and well-foundedness

Lemma. $(U, <)$ is well-founded iff the finite-chain property holds (see (6)).

Proof. The proof of this lemma is trivial.

Theorem. $(U, <)$ admits induction iff if it is well-founded.

Proof. We prove the two directions separately. First, assume that $(U, <)$ is well-founded. We shall prove that

$$(1.0) \quad \forall(y :: P.y \vee \exists(x : x < y : \neg P.x)) \Rightarrow \forall(x :: P.x)$$

Let P be a predicate on U that satisfies the antecedent of the implication in (1.0). Consider the set S defined as $\{x \mid \neg P.x\}$. If we establish that S is empty, we shall have proved one direction of the theorem.

Suppose that S is non-empty. Since $(U, <)$ is well-founded, S has a minimal element u (say). We show a contradiction, which proves that the assumption that S is not empty is false. Since u is a minimal element of S , every element of U that is less than u cannot be in S ; i.e. every such element satisfies P . Then, by our assumption about P , it follows that u satisfies P . In other words, u is not in S , which is the desired contradiction.

For the other direction, we assume that $(U, <)$ admits induction (i.e. (1.0) holds for all P) and show that it is well-founded. We assert (without formal proof) that the following is a tautology: either $DCF.y$ holds or there exists an x such that $x < y$ and $\neg DCF.x$ holds:

$$(1.1) \quad \forall(y :: DCF.y \vee \exists(x : x < y : \neg DCF.x))$$

Using the induction principle, we conclude by mathematical induction that $\forall(x :: DCF.x)$ holds. By the lemma, $(U, <)$ is well-founded.

Figure 1. Alternative proof of induction and well-foundedness

My colleague's proof is given in Fig. 1, and I invite you to compare it with the proof in Fig. 0. Note that both use the trick of equating a property with the set of values that do not satisfy it. But one proof is five times the length of the other. One requires two separate proofs; the other doesn't. One requires the fact that a well-founded set satisfies the finite-chain property; the other doesn't. One requires a proof by contradiction; the other doesn't.

More time spent comparing proofs in this fashion would give the student an appreciation for simple proofs, as well as a sense of discrimination when reading others' proofs.

The proof that well-foundedness is equivalent to the finite-chain property is interesting in its own right. Fig. 2 contains a version of the proof given by Dijkstra in [5]. This proof is not completely formal, because the property $DCF.x$ has not been formalized. Formalizing DCF isn't necessary—and indeed would just introduce unnecessary clutter. It is this search for the right blend of formalism and informality that makes mathematics, as well as programming, an art as well as a science.

We prove

$$(2.0) \quad '(U, <) \text{ well-founded}' \equiv \forall(y :: DCF.y)$$

Assume the lefthand side is true. Since $(U, <)$ is well-founded, it admits induction and the righthand side is equivalent to

$$\forall(y :: DCF.y \vee \exists(x : x < y : \neg DCF.x))$$

which is evidently true.

Now assume the lefthand side of (2.0) is false. Then there exists a nonempty set S with no minimal element, i.e.

$$\forall(y : y \in S : \exists(x : x < y : x \in S))$$

Hence, for all y , y in the nonempty set S , $\neg DCF.y$ holds, and the righthand side of (2.0) is false.

Figure 2. Proof of equivalence of induction and the finite-chain property

4. Developing correct programs

This topic has received much attention in the past ten years or so. The methods seem worthwhile and are spreading, so I won't explain much here. I am speaking of the notions of developing a program hand-in-hand with its proof of correctness, which has its roots in Hoare's work [6], was created by Dijkstra in the middle 1970s [7], was further publicized by myself [8], and is becoming traditional enough for other texts to discuss it (e.g. [3]). Martin Rem's column on algorithms in *Science of Computer Programming* is a good place to turn to for more examples and discussions of the methods. Much of our requests for more rigor and for an agility with the predicate calculus stem from the kinds of formal manipulations one does when developing program and proof hand-in-hand.

The methods for developing proof and program hand-in-hand are usually attacked claiming that the amount of detail in a large program makes formalization infeasible: 'You can't expect us to prove every single subroutine and interface correct.'

The reply to this has several pieces. First, as Hoare has said, within every big program is a little program trying to get out, and the methods proposed often let this little program out. Thus, the big programs turn out to be smaller than we thought. Second, the attempts at formalization often lead to simplification, generalization, and better understanding, giving a cleaner product that reduces significantly the time required in debugging and validation. Therefore, extra time spent in the initial design and programming may well be saved at a later period of the project. Third, suitable use of abstraction will reduce the amount of detail that has to be considered at any one time. Fourth, it is not the case that *everything* must be formalized. Look at the proof in Fig. 2; the definition of DCF_x has not been formalized — indeed, its formalization would have been counterproductive. In the same way, in a program and its proof we must learn to formalize exactly the right parts, and no more. It is this necessity to find the right balance between formality and precise informality that makes programming an art as well as a science.

Please don't misunderstand me; I am not saying that every large program can easily be proved correct. Indeed, we have little experience with the methods on large programs, and relatively few people apply formal proof methods in developing even small programs. And I am not saying that I have complete control over my own programming habits. Nevertheless, enough experience has been gained that those who know the method well feel that it belongs in the toolkit of every professional programmer.

I give here a partial development of an algorithm specified earlier because it illustrates so nicely the use of

formal syntactic manipulation in developing and presenting an algorithm. If you have difficulty with it, ask yourself whether the difficulty is with your unfamiliarity with the methods and notation or with presentation itself.

Recall the specification of the program for finding the sum of maximum-sum segment of an array. Given is an array b . ($0..n-1$), where $n \geq 0$. Desired is an algorithm that stores a value in m to establish

$$R: m = \text{MAX}(i, j: 0 \leq i \leq j \leq n: S_{i,j})$$

where $S_{i,j}$ is defined by

$$S_{i,j} = \Sigma(k: i \leq k < j: b.k)$$

Assuming that a loop will be used, we develop a first cut $P0 \wedge P1$ at a loop invariant by replacing variable n in R by a fresh variable k :

$$P0: 0 \leq k \leq n$$

$$P1: m = \text{MAX}(i, j: 0 \leq i \leq j \leq k: S_{i,j})$$

Using the bound function $n-k$, we arrive at the loop

$$k, m := 0, 0;$$

$$\text{do } k \neq n \rightarrow$$

$$\quad \text{Establish } P1[k:=k+1];$$

$$\quad k := k+1$$

od

It remains to determine how to establish $P1[k:=k+1]$ given $P0$, $P1$, and $k \neq n$. To do this, we rewrite $P1[k:=k+1]$ so that it has a term that resembles $P1$, using a range-splitting rule:

$$P1[k:=k+1]$$

$$\equiv m = \text{MAX}(i, j: 0 \leq i \leq j \leq k+1: S_{i,j})$$

$$\equiv \{\text{Split the range } 0 \leq i \leq j \leq k+1\}$$

$$m = \text{MAX}(i, j: 0 \leq i \leq j \leq k: S_{i,j})$$

$$\quad \text{max } \text{MAX}(i, j: 0 \leq i \leq j = k+1: S_{i,j})$$

$$\equiv \{\text{Eliminate bound variable } j \text{ in the second term}\}$$

$$m = \text{MAX}(i, j: 0 \leq i \leq j \leq k: S_{i,j})$$

$$\quad \text{max } \text{MAX}(i: 0 \leq i \leq k+1: S_{i,k+1})$$

Because $P1$ is initially true, this can be established by

$$m := m \text{ max } \text{MAX}(i: 0 \leq i \leq k+1: S_{i,k+1}).$$

However, the second operand of the infix max operator takes time $O(k)$ to evaluate, and we look for ways of strengthening the loop invariant to make this calculation more efficient. Introduce a new variable c with definition

$$P2: c = \text{MAX}(i: 0 \leq i \leq k: S_{i,k})$$

The range of i is $0 \leq i \leq k$ instead of $0 \leq i \leq k+1$ so that m and c are defined in terms of the same segment of array b . Some further just-as-simple calculations, which we leave to the reader, lead to the algorithm

```

k, m, c := 0, 0, 0;
{invariant: P0 ∧ P1 ∧ P2}
do k ≠ n →
  Establish (P1 ∧ P2)[k := k+1]:
    c := (c+b.k) max 0;
    m := m max c;
    k := k+1
od

```

A beautiful, linear in n , algorithm results. Unfortunately, presenting this algorithm *always* has the difficulty that the audience is not familiar with the rules used to manipulate the formulas; they have not internalized them, and therefore cannot follow the presentation easily and cannot believe that someone could use such a method. Lack of education hinders their understanding.

5. The hazards of examples

An example can certainly be worthwhile, for instance if it is used as a redundant piece of information to help build the reader's confidence in his understanding of a concept, definition, theorem, algorithm, etc. And the younger the student (intellectually speaking) the more examples may be needed to build this confidence and expertise. However, too often the example is used as the major method of explanation, essentially as a crutch by the *writer* to eliminate the need (so (s)he thinks) for a clear, rigorous, explanation. Thus, an example is used as a substitute for the specification, or an example of 'stepping through' execution of a loop is given as the only explanation provided for a loop, or a complete algorithm is explained only in terms of an example.

A brief illustration will suffice to show the state of our textbooks. A major text in data structures, which is good in many ways, gives the following sentence as the *only* explanation of a certain algorithm: 'The reader should try this algorithm out on at least 3 examples: the empty list, and lists of length 1 and 2, to convince himself that he understands the mechanism.'

This text was using a technique that I call 'programming by example': the reliance on an example (or two) for insight when developing or presenting an algorithm. This practice is less prevalent today than ten years ago, but it is still used far too often.

Let me give another illustration of this practice from the literature. Article [0] describes a neat algorithm for finding the minimum number of editing operations needed to change a given sequence of characters A into a given sequence of characters B . Two kinds of operations are allowed: delete a character from A and insert into A a character from B . The paraphrased description begins with (I have changed variable names):

Let $m.(i, j)$ be the edit distance between $A.(0..i-1)$ and $B.(0..j-1)$. $m.(i, j)$ makes sense even when i or j is zero; These values are arranged as a matrix with $1+\#A$ rows and $1+\#B$ columns. For example, if $A=abcabba$ and $B=cbabac$, the matrix of edit distances is

(7)	0	1	2	3	4	5	6	
	1	2	3	2	3	4	5	a
	2	3	2	3	2	3	4	b
	3	2	3	4	3	4	3	c
	4	3	4	3	4	3	4	a
	5	4	3	4	3	4	5	b
	6	5	4	5	4	5	6	b
	7	6	5	4	5	4	5	a
		c	b	a	b	a	c	

In this example, the entry $m.5.4$, which lies at the intersection of rows 5 and column 4 (row and column numbers start with 0) is the edit distance between $abcab$ and $cbab$. The value in that position is 3 because $abcab$ can be transformed into $cbab$ by deleting the leading a and c from the first string and then inserting a c at the front, but there is no shorter edit script for the transformation.

The paper proceeds to describe properties of such edit matrices m solely with reference to this example. In fact, the *whole description* of the algorithm is based example (7). Nowhere is there a proof of the properties of m ; they must be inferred from (7). There is a 'proof of correctness' of the algorithm at the end of the paper, but it is couched in vague terms, without a definition of m , and is opaque.

Can you conceive of a mathematics paper that deals with concepts presented solely by example? Why do we have to put up with it in computer science? When will editors and referees institute publication standards that eliminate this practice?

Actually, the definition of m and proofs of its properties are rather simple. I maintain that the minimum number $m.r.c$ of editing operations to transform $A.(0..r-1)$ into $B.(0..c-1)$ is given by $m.i.j$ defined by

```

(8) m.r.c =
  if r=0 → c
  [] c=0 → r
  [] r>0 ∧ c>0 ∧ A(r-1)=B(c-1)
    → m(r-1, c-1)
  [] r>0 ∧ c>0 ∧ A(r-1)≠B(c-1)
    → 1 + min(m(r-1, c), m(r, c-1))
  fi

```

Let us discuss this definition carefully; once we agree that it does indeed define the minimum number of editing operations we can forget about its interpretation and *work*

solely with the definition, thus placing the description of the algorithm on a rigorous and precise foundation. It is (almost) always better to translate the informal ideas as soon possible into a precise, formal form and thereafter to forget about the informal ideas and work solely with their formal definition.

The first two lines are obvious; to transform $A(0..r-1)$ into $B(0..c-1)$ requires inserting the first c characters of B (that's c editing operations), and to transform $A(0..r-1)$ into $B(0..r-1)$ requires deleting all characters of $A(0..r-1)$ (r editing operations).

Now consider $m(r, c)$ for $r, c \neq 0$. If $A(r-1) = B(c-1)$, transforming $A(0..r-1)$ into $B(0..c-1)$ is the same as transforming $A(0..r-2)$ into $B(0..c-2)$, as defined on the third line of definition (8). If $A(r-1) \neq B(c-1)$, then one can either

- transform $A(0..r-2)$ into $B(0..c-1)$ and delete $A(r-1)$, or
- transform $A(0..r-1)$ into $B(0..c-2)$ and append $B(c-1)$.

In either case, the number of editing steps is thus given as on the fourth line of definition (8).

One of the properties of m that the reader is supposed to glean from example (7) is that adjacent values differ by at most 1 and that diagonals are non-decreasing and increase by at most 2. The proof of this property, done using syntactic proof methods, gives far more confidence in the property than does example (7) (see [1]).

As mentioned above, a valid use of the example is to provide a redundancy so that the inexperienced reader can gain a measure of confidence in his understanding. However, the inexperienced reader is often likely to use the example as the major basis for understanding, and this is a dangerous practice. As an illustration of this, I offer the following.

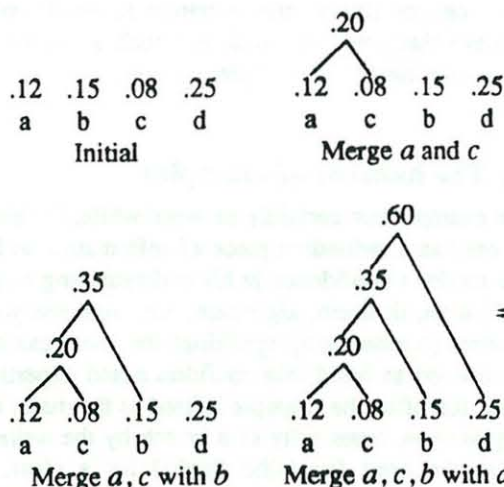
In a course on data structures, Huffman's algorithm for constructing a binary tree with certain properties from a list of (at least one) real numbers was being discussed. Consider each real number to be a tree with a single node whose value is the real number; the initial set S of trees is iteratively changed into a set containing only one tree by the algorithm below, given in my notation. I don't state precisely the task of the algorithm, except to build a tree, for that is not germane to this discussion.

Conventional set notation is used, with $\#S$ denoting the size of set S . A tree is considered to be a triple (root value, left subtree, right subtree), with $s.root$ being used to refer to the first component of tree s —the value of the root node. Function $MIN.S$ yields the tree in S that has the minimum root value.

```
do #S > 1 → var x := MIN.S; S := S - {x};
           var y := MIN.S; S := S - {y};
           S := S ∪ {(x.root+y.root, x, y)}
od .
```

Thus, at each iteration two trees are removed from S and a new tree (with the two removed trees as subtrees) is inserted. The algorithm was not presented as concisely as this in the text being used; it took eleven lines of Pascal-English.

In order to help the student understand the algorithm, an example like the following was presented in the text:



Now, a not-inconsequential number of students looked at the example rather than the algorithm and felt they understood the algorithm, especially since the instructor used exactly the same example while discussing the algorithm in class. However, the example gave the wrong impression because it did not treat a general enough case. Given the task of executing the algorithm on other data, their trees *always* had the shape shown in the above example (a right subtree is always a leaf), and this simply isn't the case with this algorithm.

A combination of things led to this situation: the students were too lazy or hadn't been taught to study the algorithm itself, the text presented an unfortunate example, and the instructor reinforced the lack of understanding by presenting the same example in class.

The lesson to be learned is that examples can indeed be the cause of a problem. The student should be warned about the dangers of examples and forced to study an algorithm in terms of its proof and not in terms of examples of its execution sequences. Teach the student to use the example as a redundant piece of information to lend assurance, but not as a replacement for a full understanding by way of proof.

6. Further comments on notation

I have made various comments on notation throughout this lecture —on the use of \supset for implication, on notations for expressing quantification, and the like. I don't think it is realized strongly enough how the notation we use colors our thoughts and habits. Of course, we have heard that restricting oneself to older versions of Fortran severely restricts the possibilities of algorithmic expression, and we believe that a programmer should know several programming languages simply to expand his notational and conceptual horizons. Nevertheless, even the most innocuous-looking choice of notation can have severe consequences.

Last year, I had the chance to work with a computing scientist whose field was algorithms, a really top-notch person. When sketching segments of algorithms in Pascal, he would invariably use a repeat loop instead of a while loop, and almost as invariably there would be a mistake —the segment would not work properly for the empty segment of an array or some similar thing. After a while, I asked him why he continued to use the repeat loop when it so often led to errors and when its proof rule was so much more complicated than that of the while loop. He replied, rather sheepishly, that, yes, he knew the while loop was typically a better choice, but the while loop always required a begin and end while the repeat loop did not (since the keywords repeat and until act as delimiters of the loop). Furthermore, the prettyprinter that he used always put the keywords on different lines:

```
while B do
  begin
    body
  end ,
```

thus making his programs ever so much longer. And on a workstation screen his effectiveness in reading a program depended on how many lines he could see at one time.

Thus, a seemingly innocuous decision about syntax in Pascal, together with a rather stupid decision by the prettyprinter designer to make programs appear as long as possible, severely hindered this scientist's work.

It behooves us to make students aware of the impact on notation, right from the beginning.

7. Conclusions

I have touched on a number of topics that I believe computer science students should be learning but are not. These topics have their factual part (e.g. the predicate calculus, a calculus for the derivation of programs), but far more than a bunch of facts is involved. The student should acquire a sense of the use of method, notation, and proof, a sense of taste, and the ability to discriminate on

technical grounds. This requires a different approach in our texts and in our teaching.

I don't mean to imply that computer scientists don't have discrimination and taste. I do believe that they don't feel these qualities are as important as I do. And I do believe that they don't realize the effect their teaching practices have on students.

I also don't want to leave the impression that I feel I have all the answers on the problem of teaching programmers. Programming —and the teaching— is a difficult intellectual task, and I feel I am just beginning to learn enough to do it well. Nevertheless, I hope that my arguments and examples will help persuade the field that change is needed.

8. Acknowledgements

My debt to Edsger W. Dijkstra and his ideas will be obvious to all who are familiar with his work; his influence can be seen throughout this manuscript. Thanks go to Prakash Panangaden and Fred Schneider for many comments on various drafts of this manuscript.

9. References

- [0] Miller, W., and E.W. Myers. A file comparison algorithm. *Software—Practice and Experience* 15 (11) (November 1985), 1025-1040.
- [1] Gries, D., and W. Burkhardt. A more rigorous description of an algorithm for finding the minimum edit distance between two sequences. Tech. Rpt., August 1987, Computer Science Department, Cornell University.
- [2] Dijkstra, E.W., and Feijen, W.H.J. *Een Methode van Programmeren*. Academic Service, Gravenhage, The Netherlands, 1984. (Also translated into German under the title *Methodik des Programmierens*, Addison-Wesley, Germany, 1985.)
- [3] Backhouse, R.C. *Program Construction and Verification*. Prentice Hall International, London, 1986.
- [4] Dijkstra, E.W. Largely on nomenclature. EWD 768, March 1981.
- [5] Mathematical induction and computing science. EWD819, April 1982.
- [6] Hoare, C.A.R. An axiomatic basis for computer programming. *CACM* 12 (October 1969), 576-580, 583.
- [7] Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, New Jersey, 1976.
- [8] Gries, D. *The Science of Programming*. Springer Verlag, New York, 1981.
- [9] Constable, R., et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.

DISCUSSION (during first lecture)

Dr. Schneider wanted to clarify whether or not the distinction between Hilbert-style and equation-style proofs is that one requires that each line follows from the previous line and the other does not. Dr. Gries replied that it doesn't matter; it is easy to devise a system for introducing new rules either way.

Professor Hoare asked if it was being suggested that equalities between expressions always have values. He noted that cases such as division by 0 must be considered.

Dr. Gries indicated that he was not worried about partial functions; Professor Hoare concurred. Dr. Gries indicated that if one side of an equation is undefined then so is the other; but if there are no free occurrences of Undefined in the expression then the result is all right even if the range is Bottom. Professor Backhouse's opinion was solicited; he indicated that he saw no difficulties in this area, but vis a vis terminology he noted that what was being called range splitting he would call range disjunction.

Professor van Westrhenen queried the formal basis of induction as described, noting that it depends upon the kind of ordering in the set. Dr. Gries responded that he had not indicated when the rule is valid, but agreed that it is certainly desirable to make the ordering explicit and to state when it is well-founded.

DISCUSSION (at end of first lecture)

It was noted that the problem specification as presented did not indicate whether or not the empty sequence was allowed. Dr. Gerhard observed that a specification could only be clarified by going back to the foundations, set theory etc.; and that this is not usually done. Furthermore, the presentation relied on equality, assuming that if one side is defined then the other side is. This is not always true. Dr. Gries felt that logicians do not understand Undefined.

Professor Backhouse commented on the syntax, suggesting as a principle that when choosing notations in terms of their properties, those properties that are most important should become invisible; e.g. parentheses can be omitted given an associative operation. A strong advantage of the notation chosen here is that both associativity and symmetry are invisible.

Dr. Clarke inquired as to how the making and discharging of assumptions fit in. Dr. Gries accepted that it was not done here, but suggested that it could be done informally. Dr. Schneider worried that this would involve the unnecessary introduction of semantics, as the point of a syntactic proof is to avoid needing a model. If we get to the point that we say that "N is even" is an assumption, then we have already thought far too much; cases should be avoided as far as possible. Dr. Clarke noted that assumptions could be described in terms of implications. Professor Pnueli thought that it was simply a question of economy in writing.

Professor Pnueli also took issue with an earlier statement that no one had done a large natural deduction proof, stating that it had been done, and seemed a bit shorter than the alternative. Professor Randell found it distressing that it was still necessary to debate notation, given the present state of the field. Dr. Gries agreed, saying that if natural deduction is shorter then that is okay.

Professor Randell wanted to standardize the notations "so that everyone could understand it." Dr. Gries suggested forcing mathematicians to use computers; this was received with laughter.

Professor van Rijsbergen emphasized the context of the discussion, asking whether the formal methods being described here were for professional computer scientists or for undergraduates. Dr. Gries wanted the goal to be to give this to undergraduates as the next generation of professionals, since none of these ideas are yet taught.

Professor Randell noted that in the ideal world, a programmer can isolate the programming task from the real world, but that in practice they don't formalize to that degree and use semantics all the time. Dr. Gries thought that a balance had to be found between formalizations and common sense, but that common sense had been overemphasized in the past.

Professor Whitfield affirmed the goal of attempting to make notation simple, but observed that mathematicians don't study notations. He thought students should be educated in how to choose simpler notations. Dr. Gries agreed that students should study notation, recalling that Dijkstra thinks that computer science will pay back its debt to mathematics in that way. Dr. Schneider felt that computer science is the first time that logic has been used in substance, but that theory is inadequate for selecting between notations. He considered empirical results necessary for such choices. Dr. Gries noted that the issue is not new, as Hilbert had brought it up. Dr. Schneider rejoined that Hilbert didn't care how it was done, to which Dr. Gries responded that Hilbert did want clarity and conciseness.

Why use Logic?

- To be able to express concepts or statements clearly and unambiguously.
- To be able to give shorter, simpler, more elegant proofs and derivations; to increase our powers of reasoning. (proofs are for human, not computer, consumption).

- An appreciation of the need for precision, rigor, elegance.
- An in-depth experience with the propositional and predicate calculi - proficiency and agility in formal manipulation.
- Appreciation of the importance of suitable, simple notation geared to formal manipulation.
- Lessening of the need for examples as a means of conveying understanding.

Complete math. induction
on $(U, <)$

$$\forall(x: x \in U: P.x) \equiv$$

$$\forall(y: y \in U: \forall(x: x < y: P.x) \Rightarrow P.y)$$

or

$$\forall(x: P.x) \equiv$$

$$\forall(y: P.y \vee \exists(x: x < y: \neg P.x)$$

$$\begin{aligned} & a \Rightarrow b \\ \equiv & b \vee \neg a \end{aligned}$$

$(U, <)$ well-founded: every nonempty subset contains a minimal element:

$$\neg \text{empty}(S) \equiv$$

$$\exists (y :: y \in S \wedge \forall (x : x < y : x \notin S))$$

Equational-reasoning

$$a + 2 = b + 3a + 2$$

$$\equiv a = b + 3a$$

$$\equiv -2a = b$$

- Substitution of equals for equals.
- Addition of same quantity to both sides.

$$\begin{aligned}
& b \Rightarrow c \\
\equiv & \neg b \vee c && \text{(Implication)} \\
\equiv & c \vee \neg b && \text{(commutativity)} \\
\equiv & \neg\neg c \vee \neg b && \text{(double neg.)} \\
\equiv & \neg c \Rightarrow \neg b && \text{(Implication)}
\end{aligned}$$

Substitution of equals
for equals.

modus ponens

transitivity of \equiv , \Rightarrow

An aside on Notation

Logicians use

\supset

for implication:

$$a \supset b$$

Let A and B be the sets of states in which a and b are true. Then

$$a \supset b \equiv A \subset B$$

Confusing!

$$\begin{aligned}
& b \Rightarrow c \\
\equiv & \{ \text{Law of Impl, Comm.} \} \\
& c \vee \neg b \\
\equiv & \{ \text{double neg} \} \\
& \neg \neg c \vee \neg b \\
\equiv & \{ \text{Law of Impl} \} \\
& \neg c \Rightarrow \neg b
\end{aligned}$$

Equational reasoning

Syntactic Proof

Natural deduction

$$\wedge\text{-I} \quad \frac{X, Y}{X \wedge Y}$$

$$\wedge\text{-E} \quad \frac{X \wedge Y}{X}, \quad \frac{X \wedge Y}{Y}$$

<u>From</u>	<u>$p \wedge q$</u>	<u>infer</u>	<u>$q \wedge p$</u>
1	$p \wedge q$	Assumption	1
2	p	$\wedge\text{-E}$, 1	
3	q	$\wedge\text{-E}$, 1	
4	$q \wedge p$	$\wedge\text{-I}$, 3, 2	

Syntactic proof

Every value of $b[i..j]$ that is not in $b[h..k]$ is in $b[h..k]$.

$$\forall (v: v \in b[i..j] \wedge v \notin b[h..k]: v \in b[h..k])$$

$$\forall (v: v \in b[i..j]: v \notin b[h..k] \Rightarrow v \in b[h..k])$$

$$\begin{aligned} & \neg X \Rightarrow X \\ \equiv & X \vee X \quad \{\text{Impl., double neg}\} \\ \equiv & X \end{aligned}$$

$$\forall (v: v \in b[i..j]: v \in b[h..k])$$

Every value in $b[i..j]$ is also in $b[h..k]$.

De Morgan:

$$0. \neg \forall(x: P: Q) \equiv \exists(x: P: \neg Q)$$

$$1. \forall(x: \text{false}: P)$$

$$2. \forall(x: P \wedge Q: R) \equiv \forall(x: P: \neg Q \vee R)$$

$$3. \forall(x: P: R) \Rightarrow \forall(x: P: Q \vee R)$$

4. Dummy renaming: y a fresh var:
$$\forall(x: P: Q) \equiv \forall(y: P_y^x: R_y^x)$$

5. One point rule: x not free in E :

$$\forall(x: x = E: P) \equiv P_E^x$$

6. Range ~~splitting~~: disjunction

$$\forall(x: P \vee Q: R) \equiv$$

$$\forall(x: P: R) \wedge \forall(x: Q: R)$$

$$\exists x \in X. P$$

$$\exists (x: x \in X: P)$$

$$\exists (x: R: P)$$

Need notation geared to
manipulative needs

$$\exists (x: R: P)$$

$$\equiv \exists (x:: P)$$

$$\forall (i: 0 \leq i < n+1: P)$$

$$\equiv \forall (i: 0 \leq i < n: P) \wedge P_n^i$$

$\{x: R: E\}$

$$\exists (x: R: P) \equiv \vee \setminus \{x: R: P\}$$

$$\forall (x: R: P) \equiv \wedge \setminus \{x: R: P\}$$

$$N(x: R: P) \equiv + \setminus \{x: R \wedge P: 1\}$$

$$\Sigma(x: R: E) \equiv + \setminus \{x: R: E\}$$

$$\Pi(x: R: E) \equiv * \setminus \{x: R: E\}$$

$$\text{MAX}(x: R: E) \equiv \text{max} \setminus \{x: R: E\}$$

BLOCK($x: R: S$)

for ($x: R: S$)

$\lambda (x: R: E)$

sequence c is a permutation
of sequence b :

$$\forall (v :: N(i: 0 \leq i < \#b: b.i = v) = \\ N(i: 0 \leq i < \#c: c.i = v))$$

Since μ is a minimal element of S , every element of U that is less than μ cannot be in S —i.e. every such element satisfies P . Then, by our assumption about P , it follows that μ satisfies P . In other words, μ is not in S , which is the desired contradiction.

For the other direction ...

$DCF.y \triangleq$ every decreasing chain beginning with y is finite.

Prove

$(U, <) \text{ well-founded} \equiv \forall(y :: DCF.y)$

If LHS is true, $(U, <)$ admits induct. and RHS \equiv

$\forall(y :: DCF.y \vee \exists(x : x < y : \neg DCF.x))$.

— If LHS false. Then there exists nonempty S with no minimal element, i.e.

$\forall(y : y \in S : \exists(x : x < y : x \in S))$

Hence, for all y in nonempty S , $\neg DCF.y$ holds

Lemma $(U, <)$ is well founded iff the finite chain property holds (every decreasing chain is finite).

Proof: trivial

Suppose $(U, <)$ is well-founded. We prove that

$$\forall(y :: P.y \vee \exists(x: x < y: \neg P.x)) \\ \Rightarrow \forall(x :: P.x)$$

Let P be a predicate that satisfies the antecedent. Consider $S = \{x \mid \neg P.x\}$. If S is empty, the theorem follows.

Suppose S is not empty. Since $(U, <)$ is well-founded, S has a minimal element u (say). We show a contradiction, which proves that the assumption that S is not empty is false.

$(U, <)$ well-founded

$$\begin{aligned} \equiv \neg \exists \text{empty}(S) &\equiv \neg \exists (y :: y \in S \wedge A(x : x < y : x \notin S)) \\ \text{empty}(S) &\equiv A(y :: y \notin S \vee \exists (x :: x < y : x \in S)) \\ &\{ P : x = x \notin S \} \end{aligned}$$

$$A(x :: P x) \equiv A(y :: P y \vee \exists (x :: x < y : \neg P x))$$

$$\equiv A(x :: P x) \equiv A(y :: A(x : x < y : P x) \Rightarrow P y)$$

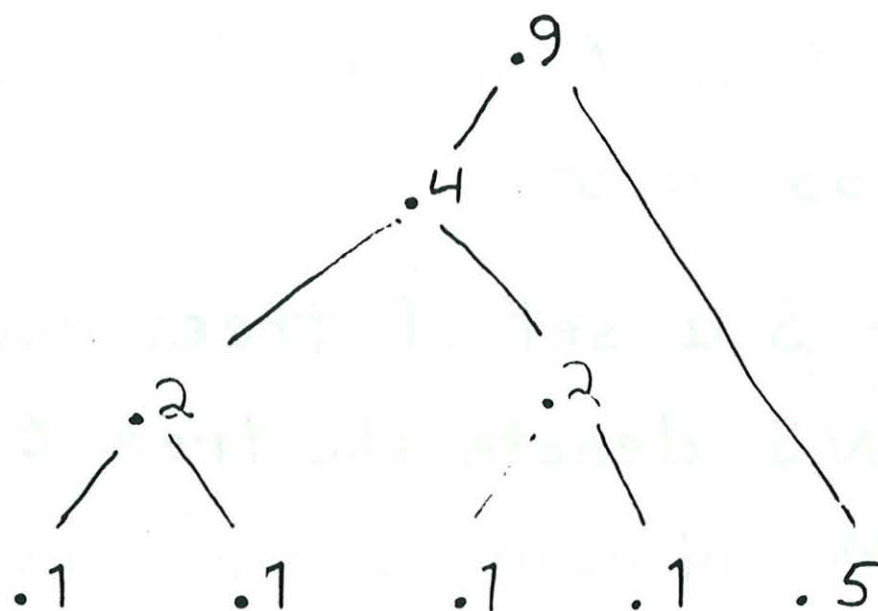
$\equiv (U, <)$ admits induction

'Try this algorithm for the empty string and for strings of 1 and two characters to be sure you understand it.'

from a modern text on data structures.

Build a tree for a given set of numbers, with the numbers as the leaves

E.g.



Let S be a set of trees, each with one node and containing one of the original numbers as root value.

A tree either is empty or consists of a triple (v, ℓ, r) where v is the root value and ℓ, r are trees.

For tree $t = (v, \ell, r)$,

$t.\text{root} = v$.

For S a set of trees, let

$\text{MIN}.S$ denote the tree t in S with minimum value $t.\text{root}$.

do #S > 1 →

var x := MIN.S; S := S - {x};

var y := MIN.S; S := S - {y};

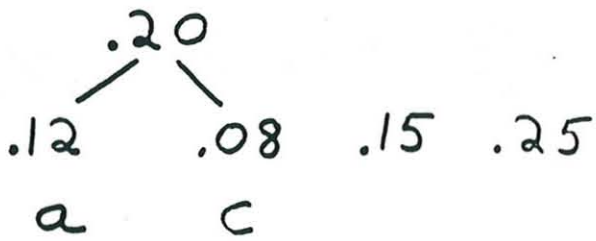
S := S ∪ {(root.x + root.y, x, y)}

od

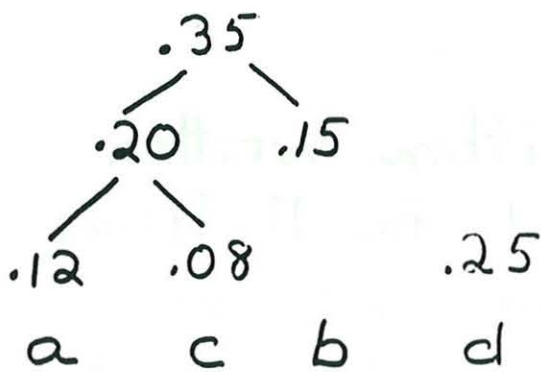
In text, algorithm written in English-Pascal, in 11 lines, and at wrong level of abstraction (pointer-implementation of trees).

.12 .15 .08 .25
a b c d

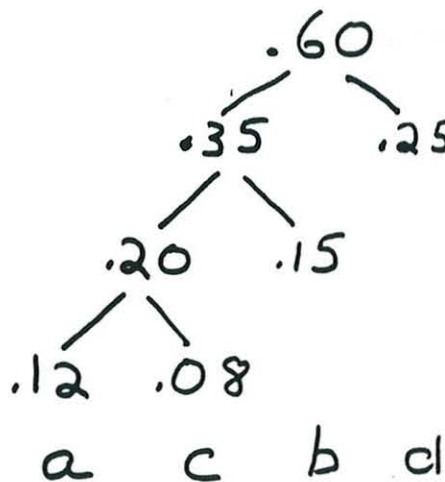
initial



merge a
and c



merge a, c
with b



merge a, c, b
with d

0	1	2	3	4	5	6	<u>A</u>
1	2	3	2	3	4	5	a
2	3	2	3	2	3	4	b
3	2	3	4	3	4	3	c
4	3	4	3	4	3	4	a
5	4	3	4	3	4	5	b
6	5	4	5	4	5	6	b
7	6	5	4	5	4	5	a
<u>B</u>	c	b	a	b	a	c	

$m[r,c]$ = min no. of editing operations
to turn $A[0..r-1]$ into $B[0..c-1]$.

$$m[r,0] = r$$

$$m[0,c] = c$$

$$m[r,c] =$$

$$\begin{cases} A[r-1] = B[c-1] \rightarrow m[r-1, c-1] \\ A[r-1] \neq B[c-1] \rightarrow \\ \quad 1 + \min(m[r-1, c], m[r, c-1]) \end{cases}$$

$b[0..n-1]$

Find the sum of the segment
(of adjacent values) whose
sum is a maximum.

$$b = (\underbrace{2, -3, 4}_{-1}, \underbrace{-1, 8}_{7})$$

Answer is 11. $(-2, -1, -4)$

$$S_{i,j} = \sum_{h: i \leq h < j} b.h$$

$$R: x = \text{MAX}(i, j: 0 \leq i \leq j \leq n: S_{i,j})$$

$$b = (-2, -1, -4)$$

Answer is

$$S_{i,j} \triangleq \sum (h: i \leq h < j: b.h)$$

pre: $0 \leq n$

post: $x = \text{MAX}(i, j: 0 \leq i \leq j \leq n: S_{i,j})$

P0: $0 \leq k \leq n$

P1: $x = \text{MAX}(i, j: 0 \leq i \leq j \leq k: S_{i,j})$

P2: $c = \text{MAX}(i: 0 \leq i \leq k: S_{i,k})$

bd: $n - k$

$k, x := 0, 0; c := 0;$

do $k \neq n \rightarrow$

$c := (c + b_k) \text{ max } 0;$

$s := s \text{ max } c;$

$k := k + 1$

od

DISCUSSION (during second lecture)

Professor Backhouse observed the use of a natural induction style proof, which he thought that Dr. Gries had said was suited more to machines than to people. Dr. Gries replied that one should not be restricted to any single method; natural induction should not be banned, but many times it is not suitable.

Dr. Gries stated that many examples are badly chosen, and that examples should only be chosen that promote understanding. Professor Wheeler wanted to know what principles could be used to select a minimal set of good examples. Dr. Gries said that students should be able to develop an algorithm by themselves with its proof; examples cannot be chosen that are good enough to teach this. Examples are only good for increasing redundancy. He remembered that he had been reading a paper and found an error; because there was no redundant information he could go no further. Professor Hoare stressed the value of counterexamples; he felt that in learning a new theory you use basic principles, as for example in group theory. These are useful in developing theorems. However, when given a doubtful statement you try to find counterexamples, that can end up motivating complexity.

DISCUSSION (at end of second lecture)

Professor Joseph inquired as to whether Dr. Gries knew of any algorithms that could not be developed in the way presented in the lecture. Dr. Gries thought that the approach in itself was often not enough; many times a key idea was needed as well.

Dr. Clarke noted that in the example recursion was avoided, and wondered if there were any strong reason for this. Dr. Gries replied that the approach used was chosen because he was familiar with it; he was sure that the problem could also be solved with recursion.

Professor Backhouse referred to Dr. Gries's statement that examples are for those intellectually young, who cannot learn immediately from a statement of principles. He observed that the algorithms are both examples of distributivity, so that $+$ distributes over Max in the summing and over Min in the editing. If operations are costed, and addition of costs is defined as multiplication, so that it no longer distributes, then the approach fails. He hypothesized that for these examples perhaps this is internalized and we look for distributivity.

Professor Rogers worried that success seemed to depend on the choice of invariant (as with Bentley), and wanted to know how such choices can be made. Dr. Gries noted that each array element had to be examined at least once; this suggests a linear approach in one direction or the other. Then, try the simplest thing first.

Dr. Holt queried the potential of the approach in giving any insight into finding parallel algorithms. Dr. Gries agreed that its usefulness in that regard was not immediately obvious.