CONSTRAINING THE CHOICE SET: LESSONS FROM THE SOFTWARE REVOLUTION

DAVID LEVY*
George Mason University

For Law, *in its true Notion, is not so much the Limitation as* the direction of a free and intelligent Agent *to his proper Interest, and prescribes no farther than is for the general Good of those under that Law. Could they be happier without it, the* Law, *as an useless thing would of it self vanish; and that ill deserves the Name of Confinement which hedges us in only from Bogs and Precipices.*

—John Locke, *Second Treatise of Government*

THE EXISTENCE OF MANMADE CONSTRAINTS on the choice set in social processes is too well known to belabor. Constitutions constrain the will of majorities; rights hamper the ability of even supermajorities to effect transactions; common morality attempts to limit our consumption decision. One frequent judgment about some of these constraints is that they are inefficient.[1] One interpretation of a state of affairs where our theory tells us an activity is inefficient, but the activity persists in spite of our valiant efforts at education, is that we really do not understand the activity. The problem of interpreting a divergence between what our theory entails and what we observe "out there" is a general one in studies of society.[2]

I shall explore the possibility that these constraints actually contribute to efficiency by examining a related self-imposed constraint: the "voluntary straight jacket"[3]accepted by those computer programmers participating in the "software revolution" or "structured programming." Using the phrase in which the case that moral constraints contribute to efficiency was originally made, the thesis to be defended below is that such constraints are employed to compensate for a "weakness" in human nature. The confines imposed in the software revolution are particularly interesting, because the normative issues

Reason Papers No. 10 (Spring 1985) 77-88.

are of an unusually simple sort: what is the lowest-cost method to attain the goal of creation of a correct program? This is a vital simplification to the argument; we need only deal with efficiency issues.[4]

One excellent reason for thinking that social institutions constraining choice exist ultimately for efficiency reasons is that David Hume said so. With Hume's argument, we can explain why sometimes we accept moral constraints by employing the same reasoning used to explain why sometimes we wear shoes: in a wide range of circumstances such artifacts reduce the costs of human activity. As Hume expressed the thesis, such institutions/artifacts exist to circumvent various human failings. We wear shoes because our skin is tender; we adopt laws to help us consider the full consequences of our actions at the moments that we perform them.[5]

When we operate within the Humean worldview, we accept the thesis that human nature has persistent characteristics that make it difficult for individual members of society to work together toward sometimes common, sometimes conflicting goals. In particular, one characteristic of ours is that as members of a species we have little concern for others as well as a rather small concern for our future self.

First, let us be clear that my interests are entirely positive. I am uninterested in prescribing behavior; rather, I consider the prescriptions that are in fact made. What devices are adopted to compensate for the social damage brought about by unconstrained individual choices? It can be shown that "moral information" which restricts the part of space individuals consider in their production decisions can enhance productivity.[6] The guidance provided by moral constraints, which would serve no purpose if offered to fully informed individuals, can serve considerable purpose if offered to ignorant ones.

There seem to be two difficulties many have had with Hume's theory of social evolution. The first problem is the slow process by which such institutions as property rights, language, and the like evolve. Even if his thesis is true, what would the relevance be of a process which takes millenia to work itself to equilibrium for those whose life span is measured in a few decades? Second, isn't the evolutionary thesis vacuous; that is, doesn't "what survives is efficient" depend upon the artful definition of "efficiency" as "survival"? To deal with the first objection, we can show that while it is true that property rights and language change only incrementally over the life of an individual, structured programming is a creature of the last two generations of computers. Even "old-fashioned," unstructured FORTRAN dates from only 1957. To come to grips with the second objection, we can show that efficiency can be given a simple enough characterization so that we can generate implications about what sort of institutions can be expected to survive.

In particular, what I shall demonstrate is that the software revolution has created imperatives requiring its adherents to renounce certain types of programming constructs, constructs that make it easier to trade future difficulties for present solutions. As Hume suggested,

positive time preference gets us into a good many difficulties. The software revolution is an evolution of institutions to get out of these tangles.

THE DOCTRINE OF THE REVOLUTION

Documentation of the claim that the structured programming revolution emphasizes the role of constraints upon behavior is completely trivial. Here is what the great theoretician E. W. Dijkstra wrote on the subject. The emphasis on constraint is clearly detailed:

> I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. If someone fears that this restriction is so severe that we cannot live with it, I can reassure him: the class of intellectually manageable programs is still sufficiently rich to contain many very realistic programs for any problems capable of algorithmic solution....
>
> Argument one is that, as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with.
>
> Argument two is that, as soon as we have decided to restrict ourselves to the subset of the intellectually manageable programs, we have achieved, once and for all, a drastic reduction of the solution space to be considered. And this argument is distinct from argument one.[7]

Earlier, Dijkstra had warned against one particular programming construct, the jump from one location to another, basing his concern on human frailty:

> Our intellectual powers are rather geared to master static relations and...our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.[8]

COMPUTER SCIENCE DISCOVERS MARGINAL COST

The Humean thesis that moral information contributes to efficiency requires, naturally enough, that we say what efficiency is. The usual definition, the minimum cost required to perform a specific task, is perfectly adequate for our purpose. Needless to say, a programming language will be used to serve many purposes and there are many resources whose cost must be considered. There is the machine time to create and run the program, the human time required to create and run the program, and so on. Machine time costs are not limited to electricity requirements; when one of my simulation experiments takes five machine hours to run, those five hours cannot be used for anything else. Other cost considerations stem from the fact that

mistakes are made in programs as well as the fact that some programs are designed to serve many purposes over decades. In either event a program will be modified, either corrected or extended, in its service life.

What will come as a surprise to economists is that the simple point made in the previous paragraph is news. Indeed, many computer scientists measured the efficiency of a program or a language implementation by the single dimension of machine time. How long does it take for the machine to run a given algorithm? For an economist it is obvious that the number of operations required for the computer to perform the algorithm is an egregiously simple-minded criterion of efficiency. Computer time is simply one input in a multidimensional minimization problem. Nonetheless, as recently as 1974, a mathematician of the stature of Donald Knuth found it necessary to point out to his peers the role of *marginal* considerations in efficiency calculations:

> There is no doubt that the "grail" of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.[10]

Knuth's "profiler," a programmer that can detect bottlenecks in programs, has had a considerable impact in thinking about language design precisely because it allows programmers to determine what parts of the code are worth further expenditure of their resources.[11]

The obvious implication of all this is that for certain problems one mix of factors will be optimal and for other problems quite a different mix of factors will be the lowest-cost method of production. Programs designed for decade-long use will put a far greater stress on the consideration of maintenance than programs designed to last a weekend.

TECHNICAL ILLUSTRATIONS

One important trade-off that confronts a language designer is the range of tasks which the language allows. By simply eliminating the possibility of performing certain activities on the machine, the difficulty with which other tasks can be performed can be decreased considerably. By the definition of efficiency then a language which cannot perform a task will not be an efficient tool for this task. Historically, languages exist with varying degrees of restriction: some languages place no restrictions whatever on use of the machine while others put very stiff restrictions indeed.

It is useful to distinguish two methods by which a computer can be controlled. The first is by means of a language which allows the pro-

grammer to coerce directly the physical machinery. To this end the language requires that the programmer specify what part of the machine is to be used for each and every operation. The most widely employed language of this type is an assembly language where there is a one-to-one correspondence between the language used by the machine itself and the language used by the programmer to issue commands. Obviously, assembly language places no constraints in the way of using the machine.[12] The second type is a language, called a high-level language, which to a greater or lesser degree conceals the hardware details from the programmer. The programmer says what is to be done, abstractly from the hardware details of how it is done. Because the machine must be addressed in its own tongue for the message to register, a high-level language must be translated (compiled, interpreted) to a lower-level language suitable for machine operation.

Machine language generated mechanically from a high-level language will generally require more computer resources (space, time) than a program originally composed in assembly language. Mechanical translation from a high-level language A to the lower level language B will, other things being equal, not result in as "tight" a piece of code as would composition in B because the translation is basically a line-by-line affair. Originally composing in assembly language can take advantage of hardware specifics.[13]

The fundamental discipline that structured programming seeks to impose upon choices is to prevent the writing of programs which are difficult to read and thus to fix or extend. There are actually good reasons such programs are written: it is easier to write poorly than it is to write lucidly. This is as true in a programming language as it is in English. As far as we know, machines do not care about style, but a poorly written program often conceals a poorly thought-through algorithm. Moreover, even a correct program which is difficult to read is often enormously difficult to modify to serve other purposes. Here, we encounter the same facet of human nature upon which Hume founds government and property: without some restraint on our self-interest we simply do not care enough for others or our future selves to act out of social concern. For the issue at hand, we simply cannot be trusted to write lucid programs without some sort of restriction on our interested actions. Since for many programming projects the costs of software maintenance (extension and correction of existing programs) dwarf the costs of program construction, it is to the interest of society for this barrier to the "quick and dirty" to arise.[14]

The first stage of the revolution was the creation of high-level programming languages for program composition. At the time of the early high-level languages (FORTRAN, and its spinoffs such as BASIC in its early years, as well as APL), computer scientists did not fully appreciate the importance of readability.[15] Consequently, many programming constructs were allowed, if not encouraged, which mask the intent of the programmer.

I have talked in generalities about language constraints. Let us con-

sider a specific problem in numerical calculation formulated in three distinct high-level languages. The only background information required for the argument is the unfortunate fact that a computer cannot do exact mathematics. A "real" number in mathematics must be represented with an infinite number of digits to the right of the decimal place. Computers, located in time and space, are only capable of representing a finite number of digits. For some purposes, an approximate answer is fine; for others, an exact answer is an absolute necessity. This means if the answer is not exact, we prefer not to have any answer. Consequently, many programming languages allow the user to specify two types of computations: integer (exact-precision) arithmetic for a very limited range of numbers and real (limited-precision) arithmetic over a very much wider range of numbers.

What is an example of such a severe approach to computational rigor? In many general-purpose languages the index of an array is an important operation that can only be performed with a number declared to be exact-precision.[16] When we want the *kth* variable in a series, we will not settle for approximately the *kth* since "approximately the *kth*" may denote the "*k–1st*" or the "*k* + *1st*".[17] Unfortunately, there is a problem of getting the two sorts of computations confused. There are three obvious ways of dealing with the problem: a) Don't allow more than one type of arithmetic; b) Make it impossible to confuse the two; c) Trust the programmer to know what he is doing.

Suppose that one wanted to use a computer to divide 1 by 2 and print the result. A long-winded version of a BASIC program to perform this otherwise intractable mathematical feat is presented below as are terse FORTRAN and Pascal programs to do the same.

```
BASIC                FORTRAN                Pascal

10 i = 1             i = 1                  program main(output);
20 j = 2             j = 2                  var i,j,c: real;
30 c = i/j           c = i/j                begin
40 print c           write(5,1),c               i:=1;
50 end           1   format(1x,f9.2)            j:=2;
                     end                        c:=i/j;
                                                writeln(c)
                                            end.
```

In the Pascal program one must first say what type of entity will be later considered. Here we specify that the variables divided are the computer realization of the (limited-precision) real numbers. The result, subject to computer precision, is 0.5. Presumably, the FORTRAN program is designed to perform the same computation. However, by naming the variables *i* and *j*, we have *implicitly* asked for integer (exact-precision) numbers. When an integer is divided by another integer, FORTRAN computes a number which is truncated towards zero. The result that will be printed out is therefore 0.00. This

confusion of real arithmetic and integer arithmetic has been the bane of many a FORTRAN program.[18] BASIC will correctly compute 0.5.

The FORTRAN program is harder to read than the Pascal program, even though it is shorter, because the programmer's intention is not made clear in the written commands. It is hard to know that the programmer did not want integer division whereas in the Pascal version this is abundantly clear.[19]

The route taken in Pascal requires that the programmer explicitly specify what variables are of what type, integer or real. Indeed, Pascal has a special symbol for integer division; FORTRAN uses the same symbol for both types of division. The route taken in BASIC is to abandon integer arithmetic.[20] Pascal programs are thus harder to write while BASIC programs cannot handle a range of problems with which either FORTRAN or Pascal programs can deal routinely.

Ultimately, the solution of ending confusion by giving up computation ability will not be satisfactory. A language cannot be efficient with respect to a problem if it cannot solve it. Thus, the great excitement generated by languages such as Ada and Modula-2 arises from their promise to deliver the same computing capability as assembly-language programming with the safety of Pascal.

LANGUAGES AS CONSTITUTIONAL RESTRICTIONS ON CHOICE SPACE

A familiar statement in constitutional theory is that an ideal constitution should be designed to serve a race of devils; angels are quite capable of taking care of themselves without laws of any sort. Just as disputes in political theory often center on the model of man assumed as background, so too controversies among proponents of different programming languages ask: what are the characteristics of the person for whom this language is designed? Here is a statement from proponents of one of the more liberal of the modern languages, C:

> Rather than try to deal with all of reality in every line of code, programming languages, explicitly or implicitly, construct models of reality and present them to the programmer.[21]

Questions of how much to trust programmers to do the right thing and how much to make it difficult to do the wrong thing are fundamental to the question of language design:

> Another model implicit in a language environment is that of the programmer. Much of the C model relies on the programmer always being right, so the task of the language is to make it easy to say what is necessary. C encourages telling the truth about strange constructions.... The converse model, which is the basis of Pascal and Ada, is that the programmer is often wrong, so the language should make it hard to say anything incorrect. In Pascal (and presumably Ada) it is harder to say strange things and therefore perhaps harder to make mistakes.[22]

A programming language provides vocabulary and a grammar in which it is possible to decide whether any particular collection of symbols is well-formed (meaningful) within that language. Distinct languages differ on the basis of what is a well-formed expression, but the fact that languages might use different symbols for the same mathematical operation is only a triviality. If one symbolic pattern performs the same syntactical function as another, a mechanical translation can turn one language into another.[23] One issue is what the language allows the program to do with the machine resources. The language provides a framework inside which instructions can be issued. Certain instructions are constitutional (of course they may be stupid), but others are not; that is, they are not well-formed expressions within the language.

The consensus among computer scientists is that the safety of a language is almost exclusively determined by its readability. There is no mechanical method of proving most programs correct, so clarity of expression is a watchword:

> In fact, program clarity is enormously important, and to demonstrate (prove?) a program's correctness is ultimately a matter of convincing a person that the program is trustworthy. How can we approach this goal? After all, complicated tasks usually do inherently require complex algorithms, and this implies a myriad of details. And the details are the jungle in which the devil hides.[24]
>
> Besides a mathematical inclination, an exceptional good mastery of one's native tongue is the most vital asset of a competent programmer.[25]

The most damning slur exchanged in the polemics among adherents of varied programming languages is the "write-only" epithet. A write-only language, allowing a great deal of computation to be accomplished in a relatively few symbols, by its very terseness hides the intent of the programmer, both from others who read the program and possibly even from the creator when he later reads the program to modify it. The highest praise possible for a computer language these days is that it is readable.

> Ada enforces a strict programming discipline with the intention of making programs more readable,...[26]
>
> But bare Fortran is a poor language indeed for programming or for describing programs. So we have written all of our programs in a simple extension of Fortran called "Ratfor"....It is easy to read, write, and understand.[27]
>
> The lack of correspondence between textual and computational... structure [resulting from GOTO statements] is extremely detrimental to the clarity of the program and makes the task of verification much more difficult. The presence of goto's [sic] in a Pascal program is often an indication that the programmer has not yet learned "to think" in Pascal (as this is a necessary construct in other programming languages).[28]
>
> It's bad practice to bury "magic numbers" [numerical constants]... in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. Fortunately, C provides a way to avoid such magic numbers.[29]

There are not all that many computer scientists who seem willing to defend poorly written programs, so the emphasis on program clarity is not at all controversial.[30] Computer science turns nasty when a further characterization of a desirable language is offered: the language must be "manageable." One characteristic of "manageable" is that the language is small, in some objective sense.[31] "Small" of course means there are few operations which are built into the language. One subjective characteristic of manageability is that an individual can hold the whole of the language's rules in his head when he writes a program. Thus, it almost follows from this artful definition that manageable languages have a single creator. The importance of such one-man languages can hardly be overestimated because they include APL, LISP, C, Pascal, Modula-2, among others.[32] Needless to say, some languages are the products of committees, examples include ALGOL 68, PL/I, and Ada. These are, in fact, very large, very complicated languages which are very difficult to understand as wholes. Hence the controversy: if you do not fully understand the language you are using, how can you be certain the program you are writing is correct?[33] Even if the program is correct, in terms of the official syntax of the language, because a big language may be very difficult to implement, what reason is there to believe the complier will be correct?

The implication of the above is simple: there will be no tendency for a unique programming language to emerge. One language that can do everything will be too big and too complicated for any one individual to remember. Thus, we obtain Adam Smith's theorem about gains from specialization with regard to programming languages. One language will process numbers; another will process letters and an individual who needs to do both will use a separate programming tool for each task.

CONCLUSION

A full-dress social institution, complete with manuals of decorum and inspiration, now exists.[34] The software revolution provides evidence that social evolution can move remarkably rapidly when there are strong enough incentives to do so. The economic cost of software failure is too obvious to belabor.

Languages that enforce discipline are triumphing over languages that allow the programmer "to do his own thing."[35] As Odysseus's ropes restrain him from the Siren, so too Programmer can escape the charms of the quick-and-dirty with the constitutional fortifications which have evolved in the last two decades. Human nature seems not to have changed much since David Hume wrote. Institutions still arise to curb our natural inclinations, to direct our self-interest so our actions more nearly serve the common good.

*Thanks are due to Peter Watts for comments on an earlier version. I have also benefited from the acute comments of *Reason Papers*' referees. The research was supported by the Center for Study of Public Choice and DARPA contract MDA 903-84-K-0331. The errors and obscurities are my responsibility alone.

1. For a recent discussion of A. K. Sen's thesis that rights and efficiency are in collision, see *Utilitarianism and Beyond,* ed. Amartya Sen and Bernard Williams (Cambridge: Cambridge University Press, 1982).

2. Examples are provided in David Levy, "Rational Choice and Morality," *History of Political Economy* 14 (1982): 1–36 and "Towards a NeoAristotelian Theory of Politics," *Public Choice* 42 (1984): 39–54.

3. The phrase is attributed to Marvin Minsky to describe Pascal, what is generally agreed to be the breakthrough language of the revolution. Jerry Pournelle, "The Debate Goes On...," *Byte* 8 (August 1983): 315. Other languages, especially some unofficial versions of Algol, embodied the same design philosophy, but Pascal has become a de facto academic standard.

4. The simplification is this: perhaps some critical constraints exist because people (or whales or trees) have rights. I believe that rights can be reduced to efficiency considerations, but there is no reason to consider the possibility that structured programming exists because machines have rights. The argument that rights encapsulate efficiency considerations is sketched in my review of *Utilitarianism and Beyond*, eds. A. K. Sen and Bernard Williams, in *History of Political Economy* 16 (Winter 1984).

5. David Hume, *A Treatise of Human Nature* (Oxford: Oxford University Press, 1888), p. 480: " 'tis certain, that self-love, when it acts at its liberty, instead of engaging us to honest actions, is the source of all injustice and violence; nor can a man ever correct those vices, without correcting and restraining the *natural* movements of that appetite." Ibid., p. 537: "Here then is the origin of civil government and society. Men are not able radically to cure, either in themselves or others, that narrowness of soul, which makes them prefer the present to the remote. They cannot change their natures. All they can do is to change their situation, and render the observance of justice the immediate interest of some particular persons, and its violation their more remote." This is the central insight upon which the argument below is based: human institutions of all sort arise to reduce the costs of attaining goals desired by many members of society.

The point is made in the modern economics literature by R. H. Strotz, "Myopia and Inconsistency in Dynamic Utility Maximizing," *Review of Economic Studies* 23 (1956): 165–80.

6. The demonstration that moral information can function as an element of a metaproduction function is provided in David Levy, "Utility-Enhancing Consumption Constraints," presented at the Southern Economic Association, November 1984; and David Levy, "David Hume's Invisible Hand in the *Wealth of Nations*," *Hume Studies* (1985), forthcoming. The former deals with convex production surfaces, the latter considers nonconvexity. Gilbert Harman has asked: how do these constraints differ from any other constraint? As I see it, there are two distinguishing characteristics of moral constraints: 1) they are "soft"; that is, they can be violated; nonetheless, 2) if violated, the individual who does so feels guilty. These conditions are of course not independent: if moral constraints were "hard," guilt would be redundant.

7. Edsger W. Dijkstra, "The Humble Programmer," in *Classics in Software Engineering*, ed. Edward Nash Yourdon (New York: Yourdon Press, 1979), p. 121.

8. Edsger W. Dijkstra, "Go To Statements Considered Harmful," in *Classics in Software Engineering*, pp. 29-30. The fame of this particular letter is hard to overstate; e.g., the Pascal compiler for the IBM Personal Computer has an option which "causes each GOTO statement in the listing to be flagged with a 'considered harmful' warning." *Pascal Compiler*, IBM Personal Computer Language Series, 1981, p. 4–14. Of course, this is a sly joke, but a joke among friends.

9. Dijkstra, "The Humble Programmer," p. 116: "To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

10. Donald Knuth, "Structured Programming with go to Statements," in *Classics in Software Engineering*, p. 269.

11. E.g., Brian W. Kernighan and P. J. Plauger, *Software Tools* (Reading, Mass.: Addison-Wesley, 1976), pp. 315–16.

12. A textbook example of such (MIXAL) is found in Donald E. Knuth, *Fundamental Algorithms*, 2d ed., vol. 1 of Knuth, *The Art of Computer Programming* (Reading, Mass.: Addison-Wesley), pp. 141–60. Knuth, "Structured Programming with go to Statements," in *Classics in Software Engineering*, p. 275: "Programs in MIXAL are like programs in machine language, devoid of structure; or, more precisely, it is difficult for our eyes to perceive the program structure. . . . It is clearly better to write programs in a language that reveals the control structure, even if we are intimately conscious of the hardware at each step."

13. For examples of how translation from a high-level language to assembly-language produces swollen code, see Christopher L. Morgan and Mitchell Waite, *8086/8088 16-Bit Microprocessor Primer* (Peterborough, N.H.: Byte/McGraw-Hill, 1982), pp. 70–72.

14. Daniel McCracken, "Revolution in Programming: An Overview," in *Classics in Software Engineering*, p. 176: "Large projects in the past have had reported coding rates in the range of two or three statements per man-day. Since it would be difficult to spend more than ten minutes writing three statements, it's clear that a lot of time was being wasted, presumably debugging and recoding modules that didn't interface properly with other modules. . . . The discipline imposed by using only the three basic program structures. . . improves the performance of even the best programmers. Perhaps more important, it can greatly enhance the effectiveness of the rest of us, who are not geniuses and who sometimes program in rather sloppy ways if left to our own devices." See also, Yourdon, "Introduction," *Classics in Software Engineering*, p. 100, "quick-and-dirty patches done in the middle of the night have a way of becoming permanent, much to the dismay of the next generation of maintenance programmers."

15. The relatively new BASIC which is provided on microcomputers is a far different language than the very early version which still can be found on mainframe computers. There are very modern versions of BASIC available today which bear a close resemblance to other structured languages.

16. Some special purpose languages, e.g., Edison, only allow exact precision numbers. Per Brinch Hansen, *Programming a Personal Computer* (Englewood Cliffs, N.J.: Prentice-Hall, 1982).

17. FORTRAN 77 is an example of a very important language which allows approximate precision numbers to serve as indices to arrays. This is one of the major changes which was made in FORTRAN vis-à-vis the 1966 standard of the language.

18. In his Alan Turing Lecture, C. A. R. Hoare defends languages such as Pascal which require the programmer to specify which type of number is to be employed for each operation, citing the fact that a Mariner space probe was lost because of the confusion of integer and real arithmetic in a FORTRAN program. See "The Emperor's Old Clothes," in *Writings of the Revolution*, ed. Edward Yourdon (New York: Yourdon Press, 1982), p. 190.

Peter Watts points out that because this is such a very well-known problem with FORTRAN, some compliers catch such abuses. However, the FORTRAN 10 complier on Brookings' DEC 10 and the FORTRAN 5 on George Mason's CDC 720 gave the answer reported in the text with nary a word of warning.

19. The programmer's intent can be inferred from the "format" statement where a real-valued number is requested. Unfortunately, many FORTRAN programs, as a matter of style, separate format statements from "executable" statements.

20. Again, this is old-fashioned BASIC. Newer BASICS do allow exact-precision arithmetic.

21. Stephen C. Johnson and Brian W. Kernighan, "The C Language and Models for Systems Programming," *Byte* 8 (August 1983): 48.

22. Ibid., p. 60.

23. This is precisely what "preprocessors" do. Why bother? One programming construct may be far more readable than another even when they are mathematically equivalent. Preprocessors are especially popular with FORTRAN 66 programmers: they allow the creation of relatively readable programs while maintaining FORTRAN 66's virtues (it is a small, widely available language).

24. Niklaus Wirth, *Programming in Modula-2*, 2d ed., corrected (New York: Springer-Verlag, 1983), p. 86.

25. Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective* (New York: Springer-Verlag, 1982), p. 130.

26. Narain Gehani, *Ada: An Advanced Introduction* (Englewood Cliffs, N.J.: Prentice-Hall, 1983), p. xiii. J. G. P. Barnes, *Programming in Ada* (London: Addison-Wesley, 1982), p. 7: "Some of the key issues in Ada are [1] Readability—it is recognised that professional programs are read much more often than they are written. It is important therefore to avoid an over terse notation such as APL which although allowing a program to be written down quickly, makes it almost impossible to be read except perhaps by the original author soon after it was written."
27. Kernighan and Plauger, *Software Tools*, p. 4. My impression is that Ratfor is the most popular of all the FORTRAN preprocessors mentioned in note 23.
28. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, 2d ed. (New York: Springer-Verlag, 1975) pp. 32–33.
29. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, N.J.: Prentice-Hall, 1978), p. 12.
30. One should acknowledge the programming equivalent of the "Real men don't..." jokes, i.e., "Real programmers don't document their code: if it was hard to write, it should be hard to read."
31. Hansen, *Programming a Personal Computer*, p. 4: "To determine whether or not a programming language is small, you need only look at the language report and the compiler."
32. The creators of the first three are Kenneth Iverson, John McCarthy, and Dennis Ritchie. Both Pascal and Modula-2 were created by Niklaus Wirth. The UNIX operating system is also a one-man affair.
33. The polemics of Dijkstra against PL/I and Hoare against Ada are especially worthy of note.
34. E.g., B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style* (New York: John Wiley, 1974); E. W. Dijkstra, *A Discipline of Programming* (Englewood Cliffs, N.J.: Prentice-Hall, 1976).
35. Pournelle, "The Debate Goes On," p. 324: "Pascal has been the real success story in microcomputing. Last year more books [counting titles] were published about Pascal than about BASIC."