It is therefore impossible from the distinct nonterminal symbol ⟨program⟩ to obtain a word which contains the symbol ⟨number⟩ and from here the string which corresponds to a signed number.

We would like to point out that we do not see the possibility, of introducing ⟨number⟩ in the arithmetic expressions of ALGOL unless we accept an ambiguous grammar.

The existence of the symbol ⟨number⟩ shows, on the other hand, the legitimate desire of the authors of ALGOL to have available signed constants, in order to avoid, whenever possible, the execution of the unary operation "change of sign."

As a matter of fact, we would like to point out that ⟨fixed point constant⟩ is defined also in [3], but it is not used in the productions which relate to arithmetic expressions.

RECEIVED MAY, 1964

REFERENCES

1. FLOYD, R. W. On the nonexistence of a phrase structure grammar for ALGOL 60. *Comm. ACM 5* (Sept. 1962), 483.
2. NAUR, P. (ED.) Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1–17.
3. RABINOWITZ, I. N. Report on the algorithmic language FORTRAN II. *Comm. ACM 5* (June 1962), 327–336.
4. SCHUTZENBERGER, M. P., AND CHOMSKY, N. The algebraic theory of context-free languages. In *Computer Programming and Formal Systems*. North-Holland, 1963.

ALGOL Note

NOTE ON THE USE OF PROCEDURES

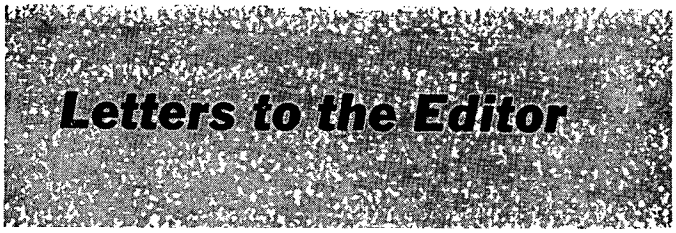
The very generality of a language like ALGOL renders it inefficient when a number of programs have to be written all dealing with a fairly narrow range of problems. This can be largely overcome by the construction of a package of suitable procedures, each of which embodies a fairly substantial piece of computation that will be required in several different contexts (see, for example [1]). A program for a specific purpose will consist of a set of these procedures linked by a more or less skeletal main program.

With this approach, some unproductive time must be spent in setting up the procedures each time they are used, establishing the required correspondences between actual and formal parameters. This militates against the use of small procedures or the placing of procedures inside inner loops, practices which may on other grounds be desirable. This drawback could be overcome very simply by introducing two new verbs such as **set** ⟨procedure heading⟩, which would set up the actual/formal parameter correspondences, and **use** ⟨procedure identifier⟩, which would activate the procedure without changing the correspondences set up by a preceding **set**. The existing procedure statement, effectively **set** followed immediately by **use**, would still be available.

REFERENCE:

1. HEALY, M. J. R., AND BOGERT, B. P. FORTRAN subroutines for time series analysis. *Comm. ACM 6*, 1 (Jan. 1963), 32–34.

M. J. R. HEALY
Rothamsted Experimental Station
Harpenden, England.



Backus Normal Form vs. Backus Naur Form

Dear Editor:

In recent years it has become customary to refer to syntax presented in the manner of the ALGOL 60 reported as "Backus Normal Form." I am not sure where this terminology originated; personally I first recall reading it in a survey article by S. Gorn [1]. Several of us working in the field have never cared for the name Backus Normal Form because it isn't a "Normal Form" in the conventional sense. A normal form usually refers to some sort of special representation which is not necessarily a canonical form; for example, it is not hard to transform any Backus Normal Form syntax so that all definitions except the definition of ⟨empty⟩ have one of the three forms

(i) ⟨A⟩ : : = ⟨B⟩ | ⟨C⟩, (ii) ⟨A⟩ : : = ⟨B⟩⟨C⟩, (iii) ⟨A⟩ : : = a.

(A syntax in which all definitions have such a form may be said to be in "Floyd Normal Form" since this point was first raised in a note by R. W. Floyd [2]. But I hasten to withdraw such a term from further use since doubtless many people have independently used this simple fact in their own work, and the point is only incidental to the main considerations of Floyd's note.)

Many people have objected to the term Backus Normal Form because it is just a new name for an old concept in linguistics: an equivalent type of syntax has been used under various other names (Chomsky type 2 grammar, simple phase structure grammar, context free grammar, etc.). There is still a reason for distinguishing between these, however, since linguists present the syntax in the form of productions while the Backus version has a quite different *form*. (It is a Form for a syntax not a Normal Form.) The five principal things which distinguish Backus form from production form are:

(i) Nonterminal symbols are distinguished from terminal letters by enclosing them in special brackets.

(ii) All alternatives for a definition are grouped together (i.e., in a production system "A → BC, A → d, A → C" would all be written instead of "⟨A⟩ : : = ⟨B⟩⟨C⟩ | d | ⟨C⟩").

(iii) The symbol ": : =" is used to separate left from right.

(iv) The symbol " | " is used to separate alternatives.

(v) Full names indicating the meaning of the strings being defined are used for nonterminal symbols.

Of these five items, (iii) is clearly irrelevant and the peculiar symbol ": : =" can be replaced by anything desired; "→" is perhaps better, to correspond more closely with productions. But (i), (ii), (iv), (v) are each important for the *explanatory power* of a syntax. It is quite difficult to fathom the significance of a language defined by productions, compared to the documentation afforded by a syntax incorporating (i), (ii), (iv), (v). (On the other hand, it is much easier to do *theoretical* manipulations using production systems and systematically *avoiding* (i), (ii), (iv), (v).) For this reason, Backus's form deserves a special distinguishing name.

Actually, however, only (i) and (ii) were really used by John

Backus when he proposed his notation; (iii), (iv), (v) are due to Peter Naur who incorporated these changes when drafting the ALGOL 60 report. Naur's additions (particularly (v)) are quite important. Furthermore, if it had not been for Naur's work in recognizing the potential of Backus's ideas and popularizing them with the ALGOL committee, Backus's work would have become virtually lost; and much of the knowledge we have today about languages and compilers would not have been acquired.

Therefore I propose that henceforth we always say *"Backus Naur Form"* instead of Backus Normal Form, when referring to such a syntax. This terminology has several advantages: (1) It gives the proper credit to both Backus and Naur; (2) It preserves the oft-used abbreviation "BNF"; (3) It does not call a Form a Normal Form.

I have been saying Backus Naur Form for about two months now and I am still quite pleased with it, so I think perhaps everyone else will enjoy this term also.

REFERENCES:

1. GORN, S. Specification languages for mechanical languages and their processors—a baker's dozen. *Comm. ACM 4* (Dec. 1961), 532–542.
2. FLOYD, R. W. Note on mathematical induction in phrase structure grammars. *Inform. Contr. 4* (1961), 353–358.

DONALD E. KNUTH
California Institute of Technology
Pasadena, California

More on Reducing Truncation Errors

Dear Editor:

In his article "Reducing Truncation Errors by Programming," *Communications of the ACM*, June 1964, Jack M. Wolfe presented a means for summing a large number of possibly small-valued terms without losing the cumulative effect of the small terms on the sum. Error would result because a floating-point variable of only eight significant decimal digits would not provide a large enough range to include the sum as well as individual addends.

An alternate way to overcome this problem, and perhaps an easier method to employ, would be to convert the addends to integers in the address portion of the computer word. This would be done by an appropriate scaling factor, plus steps similar to those first few operations in converting a floating-point number to fixed point. The addend in this form could then be added as a fixed-point variable to the summing register. When the process of summing was complete, the address integer would then be converted to a floating-point number. The scaling factor would be eliminated by division, and a more accurate sum would be achieved. With a 36-bit word, summation could reach $(2^{35} - 1)$ without overflowing, thus making this method available for ranges of ten or eleven significant decimal digits.

If it was desired to use this method where the summation might reach 3400, then addends as small as .0000001 would be included. For this example, each term to be added would be multiplied by a scaling factor of 10,000,000. For an addend variable A and a summation register IS, the following SAP instructions would be employed:

```
CLA   A
UFA   C1*
LRS   0
ANA   C2*
LLS   0
ADD   IS
STO   IS
```

When all the terms have been summed by this process, the following instructions will convert IS to a floating-point variable S, which must then be divided by the scaling factor.

```
CLA   IS
LRS   8
STQ   LS
ORA   C3*
FAD   C3*
STO   S
CLA   LS
ARS   27
ORA   C1*
FAD   C1*
FAD   S
STO   S
```

*Note: C1 = OCT 233000000000
C2 = OCT 000777777777
C3 = OCT 243000000000

RICHARD D. WHITTAKER
U. S. Navy Underwater Sound Laboratory
Fort Trumbull, New London, Connecticut

Further Comment on the MIRFAC Controversy

Dear Editor:

At the risk of belaboring the point, I would like to enter into the discussion which has been generated by publication of the article on MIRFAC [1]. Before proceeding, I admit to a programmer's bias.

First, I agree with Mr. Gawlik: something must be done to permit more widespread communication with computers by persons not trained as programmers. Second, I agree with Professor Dijkstra: English is, because of its inherent ambiguity, eminently unsuited as the means of communication [2].

It seems that Mr. Gawlik has missed the point. He ignores the discussion of the suitability of English, and he denies Professor Dijkstra's point that errors will not be uncovered by someone ignorant of programming techniques. In refuting the latter point, Mr. Gawlik uses the example of an erroneous statement by the programmer of the integrand of a Bessel function [3]. This, I am sure, is not the sort of error to which the professor referred. Errors of this type are discovered and corrected easily and quickly in the early stages of program debugging, and cause minimal time loss. It is the rather more subtle errors in thinking and logic—those which characterize everyday speech—which are so difficult to find. Any programmer can relate tales of the havoc wrought by a single erroneous bit in a program containing literally millions of correct ones.

While I do not presume to insist that one must be trained as a programmer in order to communicate effectively with a computer, it is my strong belief that one must understand the nature of the beast—and this includes a realization of the necessity for precise thinking. There is more to programming than the ability to code a mathematical expression.

REFERENCES:

1. GAWLIK, H. J. MIRFAC: a compiler based on standard mathematical notation and plain English. *Comm. ACM 6*, 9 (Sept. 1963), 545–547.
2. DIJKSTRA, E. W. Some comments on the aims of MIRFAC. *Comm. ACM 3*, 7 (Mar. 1964), 190.
3. GAWLIK, H. J. MIRFAC: A reply to Professor Dijkstra. *Comm. ACM 10*, 7 (Oct. 1964), 571.

JOHN M. SCOFIELD
IBM Corporation
White Sands Missile Range
New Mexico