CRITICAL COMMENTS ON THE PROGRAMMING LANGUAGE PASCAL*

A. N. Habermann

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

October, 1973

*This work was accomplished in the Spring of 1973 while enjoying the position of Visiting Professor at the University of Newcastle upon Tyne, England and in part was supported by National Science Foundation Grant No. GJ37728.

ABSTRACT

The programming language PASCAL is claimed to be more suitable than other languages for "teaching programming as a systematic discipline". However, an investigation of the Reports on the PASCAL language reveals that it suffers as much from ill-defined constructs as many of the languages which it is supposed to offer an alternative. Problems with the language are caused primarily by the confusion of ranges, types and structures and by the phenomena associated with goto statements.

1. Introduction

The design of the programming language PASCAL was based on the combination of two principal aims: to create "a language suitable to teach programming as a systematic discipline", but at the same time a language that can be implemented as a reliable and efficient programming system [1].

PASCAL is supposed not to contain the features and constructs of other languages that are hard to explain and are said to be an "insult to minds trained in systematic reasoning". Contrary to this statement we will see that on the one hand some useful constructs of other languages that are not hard to explain have been left out of PASCAL, whereas PASCAL, on the other hand, has features that are hard to explain and hinder the user in systematic programming.

We argue first that some useful and well understood constructs have not been incorporated in PASCAL. Secondly we go through a simple programming exercise which shows that using PASCAL as a teaching tool causes problems similar to the ones caused by using any other language. Subsequently, we discuss the major inadequacies of the language which are found in labels and goto statements, in confusing ranges, types and structures, and in procedures, functions and parameter passing. Finally, we examine the presentation of the syntax definition and the description of the semantics in the Revised Report [2].

2. Useful constructs not incorporated in PASCAL

2.1 Block structure

A sound programming principle is to declare a variable at the place where it is used. In a sorting program, for instance, a certain part of the

program can be understood as "merge two ordered sections of length p and q into one ordered section of length p+q". The merging process needs some local pointers to carry out the ordering. Programming such a sorting problem in a constructive and systematic way requires that the action of merging two sections can be written as a module that fits in an environment to which only the external specification of that module is relevant. The internal structure (to which the declaration of such pointers clearly belongs) ought to be of no concern (and definitely not accessible) to the environment. The notion of a program block as defined in ALGOL 60 [3] is a clean and well-understood construct that is very useful for this purpose.

Runtime overhead of block entry and exit is sometimes mentioned as an argument against block-structure. Such overhead, however, is very small if erratic changes of control through goto statements are not possible. Moreover, there is no need for any overhead in the absence of dynamic arrays because space for local blocks can be fixed, overlaying parallel blocks, at procedure entry.

2.2 Dynamic arrays

Changing the bounds of an array in PASCAL implies recompilation of the program. It was conjectured that a resulting gain in execution speed would more than compensate for this inconvenience. Not only is this argument very doubtful, but the implications are much farther reaching than such a statement suggests.

It is well known that execution time for accessing array elements exceeds by far the time needed for processing an array declaration. Since the former hardly depends on whether or not an array can have variable bounds, a significant gain in execution speed is not to be expected.

The true reason for not incorporating dynamic arrays in PASCAL is probably the fact that variable subranges can hardly be treated as a type.

The absence of dynamic arrays causes other inconveniences as well. Suppose we program a function LENGTH that computes the length of a vector.

```
type A = array [0..63] of real;  B = array [0..100] of real;
var p : A ; q : B ;
function LENGTH (u: ... ; n : integer) : real ;
    var sum : real ;
begin sum:= 0;
    for i:= 0 to n do sum := sum + u[i] * u[i];
    {PASCAL has no operator for exponentiation}
    LENGTH:= sqrt (sum)
end
```

The problem with the definition of function LENGTH is that we must choose between specifying the formal parameter as either type A or type B and as a result the function can operate only on one of the two types. Thus, instead of one uniform function LENGTH for all vectors, we are forced to define as many different functions LENGTH as there are vectors with different numbers of elements. The choice of the procedure statement example Transpose (a,n,m) (section 9.1.2) and of the function declaration Max (section 11) suggests by lack of any further explanation that PASCAL is in this respect as powerful as ALGOL 60. This is an unfair presentation of PASCAL's reality.

2.3 own variables

There are not many implementations of ALGOL 60 in existence that allow dynamic own arrays. If those are not implemented, storage allocation can be restricted to a mere stack discipline, whereas a heap in the ALGOL 68 sense is needed otherwise requiring considerable overhead at runtime for storage allocation and garbage collection [4]. But this is not to say that the concept of own is entirely useless. On the contrary, it serves the objective of writing well-structured programs and it can easily be defined as to allow an efficient implementation. The idea of specifying a named object as own is to make the name known only to the local environment, but in such a way that the last assigned value of the named object is retained across two successive activations of that local environment. Consider for instance the storage maintenance policy that uses the first fit algorithm as discussed in [5]. Storage consists of "free" and "used" blocks. When a request arrives for a free block of size s, the allocation agent searches for the first free block that is larger than s. Knuth observes that, if the agent starts at the beginning of the list of free blocks every time a request arrives, free blocks of small sizes tend to accumulate at the beginning of the list. But this can easily be avoided by resuming the search for a free block that is large enough at the very place where the search halted last time. The pointer that indicates this place is typically an object that should have been declared as an own variable of the allocation agent. Its value should not get lost in between two activations of the allocation agent, but the variable is of no concern to the environment in which the agent operates.

One can easily think of useful generalisations of the own concept to names that are shared by certain modules of a program, but which are inaccessible to other modules including the environment in which the former modules operate. It gives a module its private (or shared) section of global space. Observe that this own concept is basic to the structure and understanding of co-routines and concurrent processes.

Initialization of an own object is rather inconvenient in ALGOL 60. This inconvenience can easily be eliminated by incorporating the initialisation in the declaration and placing the latter as a prefix of the environment in which the own object is used.

2.4 Conditional expressions

ALGOL W has two sorts of conditional expressions, one of the form if BE then e1 else e2 and one of the form case IE of (expressionlist). It is conceivable that a teacher does not discuss these constructs when he goes through a first pass over a language with beginning programmers. But they do certainly make sense to a more advanced programmer who is concerned about a clear structure of his program. The statement

```
i := if i = 7 then 1 else i + 1
```

expresses more clearly that a value is assigned to i than the statement

```
if i = 7 then i := 1 else i := i + 1
```

in which it is more or less incidental that both alternative statements assign to i and do nothing else.

3. An exercise in programming in PASCAL

A typical problem for an introductory programming course is the sieve of Eratosthenes, an algorithm for computing the prime numbers less than a given number N. The idea of Eratosthenes' algorithm is to place the

numbers 2 to N in a row and then repeat the action of finding the leftmost number in the row followed by erasing it and all its multiples still left in the row. A prime number is found every time that the leftmost number in the row is determined.

The row of numbers 2 to N is naturally represented as an array A. Since the array bounds must be fixed, let us choose an arbitrary number for N, e.g., N = 1999. The elements of A are initialized with the value of their index. Erasing a number from the row can be implemented by assigning a zero to the corresponding element in A. Thus, a natural start of the program is:

```
begin var A : array [2 .. 1999] of integer ; i : 2 .. 1999;
      for i := 2 to 1999 do A [i] := i
```

The innocent student in programming, for instance the one who studied program structuring as presented in [6], may think that the for statement could be replaced consistently by a while or repeat statement. But an unexpected difficulty shows up if i is declared of subrange type 2 .. 1999, because

```
begin var A : array [2 .. 1999] of integer ;  i : 2 .. 1999 ;
          i := 2 ; repeat A[i] := i ; i := i + 1
                   until i > 1999
```

results in an error indication at the operator +. Section 8.1.3 is clear at this point: it requires that both operands of an addition are of type integer or real and there are reasons to assume that a phrase as "or subrange thereof" has not accidently been omitted. For, section 8.1.4. mentions subrange type explicitly in a similar place; furthermore, subrange type is not an instance of scalar type (see section 6.1); finally, a type

can hardly be associated with the result of an addition of two operands of subrange type.

It seems as if the problem can be avoided by writing i := succ(i) instead of i := i + 1. But now the test i > 1999 fails at the very moment that the repeat statement is about to terminate, because succ(i) is undefined when i = 1999 (section 11.1.4). The proper solution is to declare variable i as integer instead of as subrange type. (A clever programmer will of course use the trick of declaring i of subrange 2 .. 2000 and not use element A[2000]).

However, the use of i as index expression is strictly speaking illegal when i is declared of type integer, because the type of i does not match the index type of array A (section 6.2.1). If this were true, there is hardly a way around applying a trick as mentioned above. But the report is sufficiently vague at this point as to allow a different interpretation. The crucial phrase used in the report is that index expression and index type must "correspond" (section 7.2.1), whereas in similar situations the phrasing "same type" or "identical type" is used (6.2.1, 8, 9.1.1, 9.2.3.3). The correct interpretation of the word "correspond" seems to be that at runtime the evaluated expression must happen to be in the subrange as determined by the array type definition. It will interest advocates of compile time checks to find out that this interpretation implies at least as much checking at runtime as when ranges are not considered as types.

Our previous experience suggests that we program the search for prime numbers by means of a <u>for</u> statement.

```
for i := 2 to 1999 do
if A [i] ≠ 0 then
begin PRINT (i) ; erase all multiples of i end
```

A new difficulty arises when we program "erase all multiples of i". We would like to go through array A in steps of i, but PASCAL provides only a fixed step element of one or minus one. We can, of course, create a range that can be stepped through in steps of one and compute the index value into array A as a function of the successive elements of this range. We then get:

```
for k := 1 to 1999 div i do A[k * i] := 0
```

Programming "erase all multiples of i" this way incurs paying the price of an integer division and a multiplication that is repeatedly evaluated. We can avoid the latter at the cost of an additional variable that holds the value of the index expression. The declaration of this variable must be added to the program heading and it turns out that a subrange type cannot easily be used as type for any of the variables for which this would make sense.

A simpler program is obtained, after all, if "erase all multiples of i" is programmed as a while statement. We won't pursue, however, the details any further, because the program is not really important here. The purpose of the exercise was merely to show that a teacher who uses PASCAL cannot avoid discussion of language peculiarities just as he would when he used another programming language.

4. Labels and goto statements

It is surprising that in the design of a tutorial language the issue of programming without goto statements is totally ignored. This does not seem to be very much in the spirit of structured programming as presented in [7]. But even so, the secondary aim of PASCAL to provide a fast language system should have prevented inclusion of the goto statement because of the trouble it causes in a compiler, especially in a one pass compiler. An example of the difficulties a one pass compiler has to cope with because of labels and goto statements is sketched below.

```
procedure P ; label 1 ;
      procedure Q ;
          procedure R ; begin ---goto 1 ; ---goto 1 ; ---end {R};
      begin --- goto 1 ; --- goto 1 ; ----- 1 : end {Q} ;
begin --- goto 1 ; --- goto 1 --- 1 : end {P} ;
```

A non-local label requires a forward declaration as in procedure P. It seems as if the goto statements in procedures Q and R refer to that label. However, the label at the end of Q definitely changes the interpretation of the goto statements in Q.

At this point we may conclude that the program is in error because label 1 should have been declared as global label in the heading of procedure Q (section 10). But further scanning leads ultimately to a label defined at the end of procedure P for which a global definition certainly makes sense, so the conclusion may be that no mistake was made after all. If the goto statement should be incorporated, it probably ought to be restricted to local labels. A separate provision can be made for jumping to error handling procedures that cause an automatic change of scope.

The Revised Report is sometimes vague and probably mistaken in other places about labels and the consequences of goto statements. First, it is doubtful whether or not a label in front of the statement part of a procedure declaration is considered as "in the procedure" or not (see 9.1.3). We assume it is, because otherwise the problem arises that control could be transferred to such a labelled statement without activation of the procedure. Second, in the Revised Report, the scope of a label is defined to be the procedure within which it is defined (section 9.1.3). We assume that it is a mistake that functions are not mentioned in the scope rule for labels, because it seems at least as strange to jump into a function as into a procedure. Finally, the change in scope definition from compound statement, as in the original Report, to procedure and the absence of block structure together cause the notorious problem of jumping into a for statement. There is nothing in PASCAL that prevents this and it seems hard to impose this restriction gracefully given the definition of PASCAL.

The Revised Report resolves the ambiguity of labels and case labels as it existed in the original Report by using comma as separator between case labels, by using colon as separator between the rightmost case label and the statement label, and by restricting the number of statement labels to one.
The statement

```
4 : case i of
   1, 2 : 3 : goto 3 ;
      4 : goto 4 ;
   5 : 6 : goto 5 ;
   6 : 5 : goto 6
   end
```

is then correct according to the Revised Report, but realize that the first alternative is the only one that, once selected, repeats merely itself.

5. Subranges, types and structures

The most unsatisfactory aspect of the PASCAL language is the artificial unification of subranges, types and structures. This has a negative effect on the tutorial qualities of the language, it conveys a narrow view on types as merely ordered sets of values and it causes problems for the programmer as will be shown below.

It turns out that subranges cannot consistently be treated as types and vice-versa. E.g., using scalar types as subranges legalizes the declaration

```
var A : array [real] of integer
```

The program exercise in the preceding section presented several examples of the difficulties that arise if subranges are strictly treated as types with respect to expressions, control ranges and index expressions. Such problems of interpretation are not just restricted to subranges of type integer as is shown in the example below.

```
case succ(d) of
   Tuesday, Thursday, Friday : S1 ;
      Wednesday, Saturday : S2 end ,
```

Suppose variable d is declared of subrange type workday, which is defined as subrange Monday .. Friday. The case expression succ(d) is also of type workday if we hold on to the strict interpretation, so the statement contains a type conflict because of label Saturday (section 9.2.2.2). Another question is how succ(d) should be interpreted when d = Friday, because Friday has no successor in type workday.

The idea of treating subranges as types is completely abandoned in case of assignment statements, because the type of the variable is even allowed to be a subrange of the type of the expression to be assigned (section 9.1.1). One may expect that the same rules apply to value parameters, although nothing is said about subranges in section 9.1.2.

Instead of considering subranges as types, the following rules should apply

a) the type of an object in PASCAL declared of subrange type, st, is the type of the super-range of which st is a section;

b) ranges are evaluated and tested at runtime. It would be feasible to consider PASCAL subrange declarations as type declarations with a range attribute for runtime checks.

Consider subsequently the treatment of structures as types. The PASCAL language has four fixed structuring rules indicated by the word delimiters array, record, set and file. A useful rule in PASCAL is the composition of array and record structures such as

```
type R = record vec : array [1..10] of integer end;
     A = array [1..10] of array [1..10] of R ; var s:A
```

Although the Revised Report contains only one trivial example of accessing such structures or their components (section 7.2.2), a PASCAL compiler test showed that all useful constructs are accepted on the left hand side of an assignment statement : s, s[i], s[i][j], s[i][j].vec and s[i][j].vec[k].

However, the composition rule is not enough to justify the idea of treating structures as types. It turns out that in all relevant language constructs, except assignments, structures are, or ought to be,

treated differently from simple type objects or pointers in PASCAL. One has access to elements of a structure, but (of course) not to the structure of a simple typed object or pointer. Structured objects cannot be used as operands in algebraic expressions and should not be used as index expressions. The default parameter passing rule for simple types and pointers is by value, but the default rule for structures should be by reference. Range expressions in array declarations and control statements such as for statements or case statements can be of certain simple types, but should not be structures. So, the similarity of treatment in assignment statements does apparently not carry over to any other language construct.

The notion of simple type attempts to distinguish somewhat between types and structures, but, unfortunately, structures sneak in again by means of type identifiers. The declaration v : A parses variable v as being of simple type (section 6.1), so the declaration

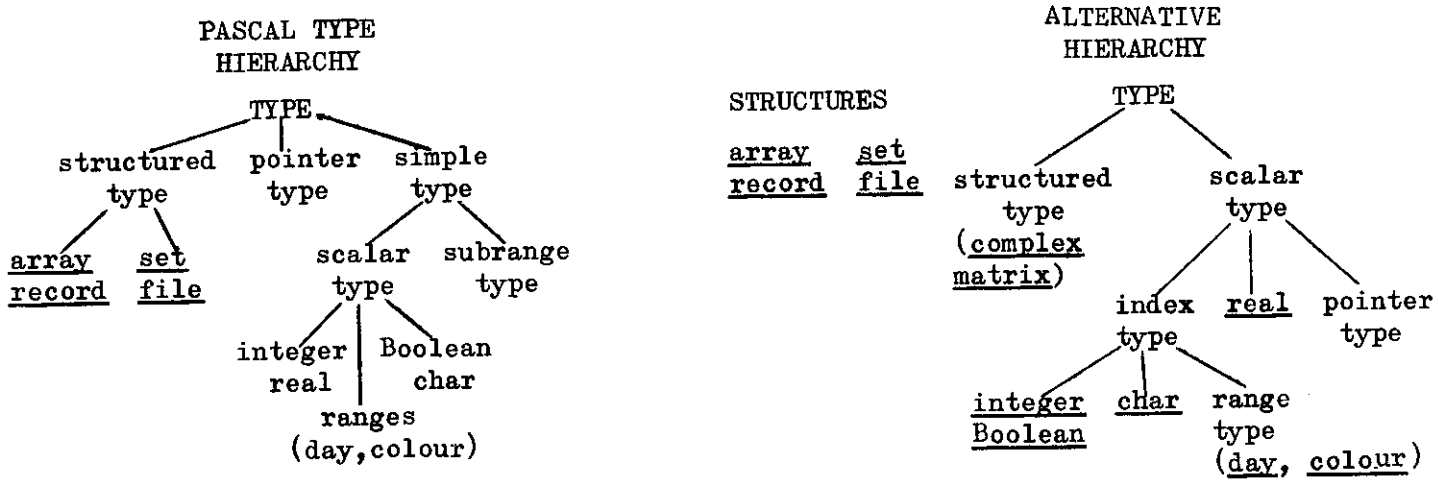
var p : array [A] of v

is legal in a procedure. Observe that this declaration is legal irrespective of how type A is defined! It could be defined as array or file or even a composition of those.

A useful distinction between types and structures is based on two principles a) a typed object is treated as an atomic entity, i.e. the type definition hides the structure of the objects, whereas elements of a structure can be accessed anywhere within the scope of existence; b) the major constituent of a structured type definition is the set of operations that can be performed

on the objects of the type, whereas changes of structures are solely accomplished through operations on the elements.

Array and record are examples of structures, matrix and complex are examples of types. The type definitions hide the detailed structure and provide operations directly on objects of type matrix or complex. The type hierarchy of PASCAL is compared with the proper type hierarchy in the diagram below.



6. Procedures, functions and parameters

In the original version of PASCAL an attempt was made to avoid side-effects in functions by means of the rule that assignments to non-local variables were not allowed. This attempt failed, of course, because a side-effect could also be caused by a procedure call or by the use of pointer variables. The restriction has therefore been left out of the Revised Report (of which fact no notice is given in section 11 in which functions are defined).

The difference between procedure and function is now so marginal that it is really not worth keeping. If it were useful to distinguish the two in the compiler in order to check whether or not an assignment to the function identifier occurs, the compiler could easily do so by means of the presence

or absence of a function type identifier in the heading. A distinction, as in BLISS, between function and routine seems much more useful, because it serves two purposes, that of improving clarity of program structure and that of efficiency during compilation and execution [8].

We argued before that the default case of passing an array, file, set or record ought to be by reference, because call by value implies that a complete copy of the structure must be passed across to the procedure or function activation.

A concept that leads to much inefficiency, particularly at runtime, is the formal procedure or function parameter. Example

```
procedure P (procedure F) ;
   var p, q, r : integer ; x, y, z : real ;
begin ---- F (x, y, z) ; --- F (p, q, z) ; --- end ;
procedure A (b, c, d : integer) ; begin --- end {A} ;
procedure B (u : real ; v : ↑R ; w : f) ;
   begin ----- end {B} ;
P (A) ; --- P (B) ; ---
```

It is hardly possible to perform all the necessary type checking at compile time and therefore code must be generated to check the types at runtime. One solution is to require full specification of the formal procedure or function with respect to its type and the types of its parameters. This would be consistent with the requirement for full specification of any other formal. It would have made much more sense if attention had been paid to

this kind of consideration and simple forms of procedures or functions than to eliminating side-effects or creating an artificial distinction between procedures and functions.

7. The Revised Report

The description of the semantics is rather inaccurate and incomplete at times. Some of the changes have been indicated, but several major revisions remain unmentioned. E.g., the scope rules for labels in section 9.1.3, the removal of the assignment restriction in functions, several type productions in section 6.

The definition of ⟨base type⟩ is an example of the inaccuracy of the Report. The semantics of section 6.2.3 describe ⟨base type⟩ as being not structured type. From the production ⟨type⟩ at the beginning of section 6 we conclude that ⟨base type⟩ apparently goes to ⟨simple type⟩ or ⟨pointer type⟩; but the production in section 6.2.3 for ⟨base type⟩ excludes ⟨pointer type⟩. And there are many more. The general experience is that one can start at an arbitrary point in the Revised Report on PASCAL and will inevitably find a little mistake or a not precisely described notion after a while. A constant has no type; yet, the definitions of subrange and case statement depend on the type of constants. What are we to think of undefined notions as "corresponding types", "operation", "outside a procedure" etc.? Right at the beginning the notation { } is introduced. Yet, when it should be applied for the first time, the superfluous symbols * and ⊕ are used. First we learn that the functions succ and pred apply to arguments of scalar type. When subranges are introduced, nothing is said about these functions in spite of the fact that subrange type is not included in scalar type (section 6.1).

Yet in section 11 we find that the functions succ and pred apply to both types. All these flaws, omissions and inconsistencies demonstrate that it may not be so easy to achieve precision and consistency in an informal description as was suggested in the introduction of the original Report (section 1, page 6).

Conclusion

The result of designing the PASCAL language is disappointing in view of the high spirits and strong statements in the introduction of the original Report. It is nice that a programmer can define types, but a type should not merely be viewed as a value range. We saw that subranges can hardly be treated as types, while structures and types do not allow a similar treatment in any language construct except, apparently, in assignments.

Paying attention to tutorial qualities of a language is laudable, but unacceptable in this regard are the confusion of subranges, types and structures, the inclusion of goto statements and the inferior presentation of the language in the Revised Report. It is worthwhile to strive for a language that can be supported by an efficient programming system, but this objective should not have led to the exclusion of some well-defined concepts present in other languages, whereas it should have resulted in better specification and substitution rules for parameters and a useful distinction between functions and procedures.

It would be regrettable if PASCAL is going to be fixed in its present state, as the introduction of the Revised Report seems to do. There are still many fundamental language design issues to be discussed in general. Among the practical points are:

grouping of statements by means of bracket pairs or control delimiters;

initialisation in declarations;

simple assignment operations of the sort "add to variable".

Among the major issues are:

type definitions as a template for structured objects (cf mode and operation definitions in ALGOL 68 [4] and the class concept in SIMULA 67 [9]);

structure definitions as a description of the access algorithm to elements of an instance of a structure;

control statements and rules for leaving scopes of control.

A small language of the PASCAL sort can, of course, provide only limited capabilities with regard to these major issues. It is therefore acceptable that PASCAL has fixed structuring rules, but viewing subranges and structures as types is a deplorable oversimplication. The value of the PASCAL design and implementation effort is in stimulating research and development of language constructs in view of the present state of the art of programming. However, the language will defeat its purpose if it is going to be consolidated in its present form with all its flaws and inconsistencies for the sake of compatability. Instead of presenting a particular language as the solace, we had better continue a discussion on language issues and analyse their impact on programming systems.

Acknowledgement: Comment by Profs. D. Gries and J. J. Horning has been most helpful to improve the presentation of this study.

References

1. Wirth, N.: The programming language Pascal. Acta Informatica 1, 35-63 (1971).

2. Wirth, N.: The programming language Pascal (Revised Report). Berichte der Fachgruppe Computer-Wissenschaften, Eidgenössische Technische Hochschule, Zürich 1972.

3. Naur, P. (ed.): Revised Report on the Algorithmic Language ALGOL 60. Comm. ACM 6, 1 - 17 (1963).

4. van Wijngaarden, A. (ed.): Report on the Algorithmic Language ALGOL 68. Numerische Mathematik 14, 79-218 (1969).

5. Knuth, D. E.: The Art of Computer Programming, Vol. 1 (ch. 2). Reading (Mass.): Addison-Wesley 1968.

6. Dijkstra, E. W.: A Short Introduction to the Art of Programming. Dept. of Mathematics, Technological University Eindhoven, EWD 316, 1971.

7. Dijkstra, E. W.: Notes on Structured Programming. In: Dahl, O. J., Dijkstra, E. W. and Hoare, C.A.R.: Structured Programming. London: Academic Press 1972.

8. Wulf, W. A. et al.: BLISS: A Language for Systems Programming. Comm. ACM 14, 780-790 (1971).

9. Dahl, O. J. et al.: Simula 67: Common Base Language. Norwegian Computing Center, University of Oslo 1967.

Professor A. N. Habermann
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
USA