Reply to a paper by A.N. Habermann on the programming language Pascal

Olivier Lecarme and Pierre Desjardins
Research group on programming systems and languages,
Département d'Informatique,
Université de Montréal,
C.P. 6128, Montréal 101,
Canada.

Summary.

A.N. Habermann recently published some "critical comments on the programming language Pascal". His reproaches are principally that numerous constructs are ill-defined, that there is "confusion" amongst ranges, types and structures, and that the *goto* statement should have been abolished. The present reply successively deals with points that are clearly refutable, those which are debatable and those which constitute valid criticism. Its principal aim is to encourage the reader to form his own opinion.

1. Introduction

The stated purpose of the paper by A.N. Habermann [1] is to demonstrate that the programming language Pascal, as defined in the Revised Report [2], is not suited to any of its objectives: it is claimed that it lacks some fundamental constructs, that it is a poor teaching tool, that it contains some totally inadequate features, and finally that it is incompletely described by a document full of errors.

The reputation of its author [3,4,5] can make the paper harmful to Pascal, so we think a reply is in order. We will attempt to correct the unfavourable impression that readers, without any knowledge of Pascal, could get from Habermann's criticisms. We will not follow his argumentation point by point, but rather classify the subject matter into four parts: clearly refutable points, points which are at least debatable, valid criticisms and finally misunderstandings and minor errors. The aim of our reply is to encourage the reader to form his own opinion, if possible by practical experience with the language, or in any case, at least by a careful reading of the basic papers relevant to Pascal ([6] or better [2], [7] and [8]).

2. Refutable points

In this section, we deal with points which are, in our opinion, clearly refutable, i.e. criticisms which resulted from a misinterpretation of the basic aims of Pascal or a misunderstanding of some major aspects of the language itself. It is possible that some people might have preferred a different distribution of points between this section and the following one.

2.1. *Useful constructs not included in Pascal*

Habermann suggests four such constructs, of very unequal importance. The main point to note is that it would be very easy to continue adding constructs to the language indefinitely: Pascal does not contain all the constructs which may be considered useful, nor even all those present in other programming languages. This is because creation of an endless list of constructs is clearly not the right direction to follow for the development of better programming languages. The most unfortunate attempt in this direction is that of PL/I [9], and even its most irreclaimable addicts and most enthusiastic eulogists always seem to find more constructs to incorporate in it [10,11].

In fact, one of the principal strengths of Pascal is that it is a simple and concise language, including only what is vital for reaching its aims. We remind the reader that there are only two of them: to allow the teaching of programming as a systematic discipline, and at the same time to be implementable in a reliable and efficient way. These objectives are precisely the most difficult ones to reach when using languages which try to incorporate all "useful constructs". The author of Pascal has therefore severely restricted the number of facilities, and it is quite sure that almost everyone will find missing certain of his favourite constructs. In section 3, we shall examine three of the "left out" constructs regretted by Habermann.

2.2. *An exercise in programming in Pascal*

The simple exercise worked out for the reader by Habermann is supposed to prove that Pascal is a poor tool for teaching programming. All that such an example demonstrates is simply that it is possible to use Pascal badly, which is of course true for any tool. If one tries to learn to write using his pencil upside down, the bad results will not be in any way the fault of the pencil. Consequently, we prefer to rework the part of the example which is given, to show that in actual Pascal no difficulties arise.

The problem is to compute prime numbers using the sieve of Eratosthenes. A comparison of the different algorithms available, even superficially, should be useful before trying to put down a solution [10,12], but this precise algorithm is not so bad, and it has been particularly well investigated by Dijkstra [13], Hoare [14] and Wirth [15,16].

Habermann chooses to represent the numbers between *2* and *n* by an array of integers, in which every element contains its own index: to remove a number from the sieve, one will assign a zero to the corresponding element. Although using the set structure of Pascal should be far better [14,15], we shall use simply a Boolean array, not only because it seems more natural but because we will encounter the same problems with indexes as Habermann did. A natural way to start-off the program is:

```
const n = 1999;
type sievesize = 2..n;
var A : array [sievesize] of Boolean;
    i : sievesize;
begin for i := 2 to n do A [i] := true
```

The constant and type declarations for *n* and *sievesize* are not mandatory but very useful: they contribute to the clarity of the program; the number *1999* textually appears in only one place; a modification of the program to deal with *3000* or *200* numbers would require modification of the constant declaration only.

Using as pretext the ideas of Dijkstra [17], Habermann then proposes to replace the *for* statement by a *repeat* statement. This modification is completely useless, since the *for* statement is perfectly adapted to situations where the number of iterations is known before entering the loop. Moreover, the program could become less efficient, and will surely be less clear. But the modification brings forth an interesting point: in a *repeat* or *while* loop simulating a *for* loop, the control variable needs to take on one more value than in the *for* loop. This is not inherent to Pascal but to the meaning of these statements. The natural solution in Pascal is to declare *i* on a subrange longer by one than the index type of *A*:

```
const n = 1999;
type sievesize = 2..n; indexsize = 1..n;
var A  :  array [sievesize] of Boolean;
    i  :  indexsize;
begin i := 1;
    repeat i := i+1; A [i] := true
    until i = n
```

Furthermore, there is no problem in using the operator +, since all operators which are defined on integer operands also accept operands whose type is a subrange of the type *integer*. This is quite obvious, otherwise, there would have been no point in defining subrange types. Furthermore, all the above is clearly stated in Axiomatic Description of Pascal [8], as we shall see in section 2.3. Similarly, there would be no problem if we choose to write *i := succ(i)* instead of *i := i+1*:as a general rule the *succ* function has no cognizance of whether or not its argument was declared to be of a scalar type or of a subrange of that type. So in the case of *i := succ(i)* when *i = 1999*, the successor value is defined since 2999 does have a successor in the base type *integer*. Finally, the fact that the index type of *A* is not the same as the type of *i* is no problem either; both have the same base type, i.e. *integer*, and the only validity condition for array references is that index values fall within array bounds, as in all programming languages.

The next section of the program deals with the search for prime numbers, and is straightforward. Add a variable *k* of type *0..n* (which can, for the sake of simplicity, serve also for *i*) then:

```
for i := 2 to n do
if A [i] then
begin write(i);
    {erase all multiples of i  :}
    k := 0;
    while k  ≤ n-i
    begin k := k + i; A [k] := false end
end
```

Habermann's example stops here, consequently so does ours. The conclusion derived by Habermann is that Pascal is no more suited to teaching programming than any other language. We have already said at the beginning of the present section that one can without proving anything, write bad programs in any language. What seems more serious to us is that this section, as well as the remainder of Habermann's paper, places criticisms of the style of the Report, and criticisms of the language itself, on the same level, and incorrect interpretations which may result from the former are used to try to belittle the latter, by systematically using its possibilities at the wrong time.

2.3. *Subranges and types*

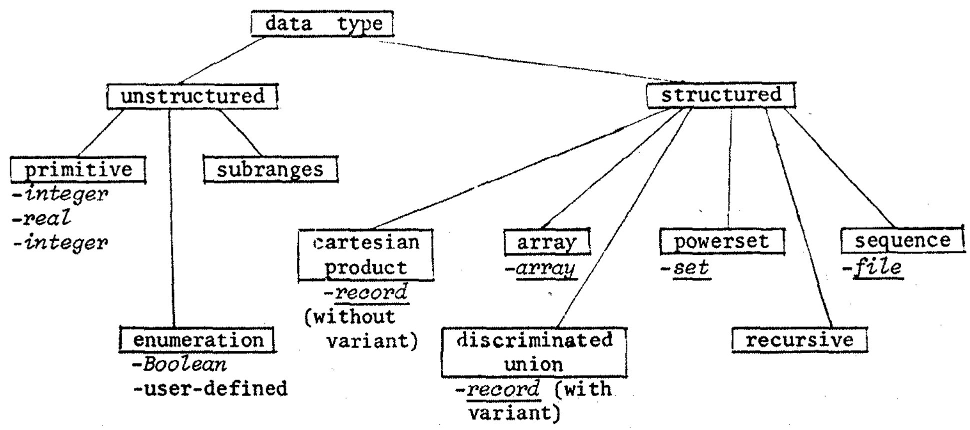
One of the most original aspects of Pascal is the whole notion of type. To use the same terminology as Habermann, this one notion unifies different concepts which may be named "type" (the manner in which bit patterns must be interpreted),"range" (the set of possible values for a variable of the given type) and "structure" (a template for storing data). We shall defer until section 3.5 the debatable parts of this novel approach, and examine here only the clearly positive points.

One must not fear that declaring a variable of subrange type makes that variable lose all the properties of the base type from which the subrange is taken. On the contrary, this variable inherits all the properties of the base type, plus the possibility of having run-time checks performed whenever values are assigned to that variable and also the property of possibly taking up less space in memory. Another most important point to understand is that the type of an expression is not always the same as the types of variables in it. This is true even in Fortran or Algol 60 : for example, in Algol 60, *1/2* is of type *real* while its operands are of type *integer*, and $1 < 2$ is of type *Boolean*. The Axiomatic Description of Pascal is perfectly clear on the matter. Given a scalar type *T* and a subrange type *S* extracted from *T*, if *a* and *b* are variables of type *S* and ⊗ an operator defined on type *T*, then the expression $a \otimes b$ is legal, and yields a result of type *T*. The assignment statement is handled in a similar way, but assignment of a value of type *T* to a variable of type *S* is not always legal.

Of course it is true that most of the validity checks involved in subrange types must be made at run time, but it is an easy thing to have them performed only when the user requests so (or better still, always have them made unless the user explicitly says otherwise); moreover, they constitute an invaluable debugging aid.

Subranges have other important qualities. Their mnemonic and descriptive value is such that a well-written Pascal program generally does not contain any integer variables : they are replaced by variables whose type is a subrange of type integer. Another important aspect is that they allow the user to control the space occupied by a variable of subrange type, for example when he includes one as an element of a packed structured type; this is much more general and machine-independent that the *long* or *short* attributes of *real* and *int* in Algol 68 [18].

Section 5 of Habermann's paper concludes with a tree diagram supposed to represent the type hierarchy in Pascal. A more correct version of this diagram follows. It uses Hoare's terminology [19], since Pascal implements most of the ideas presented in this paper. One can see that the type "pointer" is not explicitely named, since in Pascal a pointer variable always designates a value of a given type; pointers do not constitute a true type, but are only a method used to implement recursive data type [20].



2.4. *Miscellaneous*

In section 2.2, Habermann writes in an example the expression $u[i] * u[i]$, and adds as comment that Pascal has no operator for exponentiation. For this precise case, Pascal gives the function *sqr(x)*, which squares its argument *(real, integer* or subrange thereof*)*. For most compilers, the generated code for a multiplication would be better than the one for $u[i] \uparrow 2$, which would probably require evaluation of a logarithm and an exponential. More generally, the exponentiation operator was not made a part of Pascal for the sake of simplicity and clarity. If one tries to completely describe it, for all valid and invalid combinations of operand types, signs and precisions, one inevitably obtains several pages of complicated explanations and tables [21].

Another clearly refutable suggestion is the one which is made in section 6, to have a default passing mode for structured values in procedure calls. Such a proposal would only introduce an incoherent particular case into the language, making programs less clear. The example of PL/I clearly shows the danger of default options which depend on context and on the nature of objects, especially in parameter passing. One of the basic principles of Pascal is to hide nothing from its user, and to do nothing in his place, as would be the case if a parameter was supposed to be variable simply because it was a structured one.

3. Debatable points

In this section, we deal with points on which reasonable people can disagree in all honesty. Generally, the direction chosen in Pascal is clearly not the only reasonable one, and a different approach would surely have its qualities. However, the solutions taken for Pascal generally fit well in the whole philosophy of the language, especially as to clarity and simplicity.

3.1. *Block structure*

Pascal does not provide a block structure in exactly the same sense as Algol 60 [22], since all declarations are made at the procedure level, the program itself being a degenerate procedure. Therefore, it is not possible to open a block in the middle of another, simply by introducing some declarations after the *begin* symbol. However, it is very important to clearly distinguish between the different possibilities given by the block structure of Algol 60, and to see what possibilities Pascal lacks because of its different approach. Both languages provide dynamic storage allocation of variables, as well as the notion of locality of declarations. Only Algol 60 offers the possibility of including two disjoint blocks within a single other one, thus saving storage by assigning variables of both blocks to the same area. In Pascal, this economy is only possible at the procedure level.

The advantage of the approach taken in Pascal is, once more, that of simplicity. Declarations are clearly separated from instructions, being grouped between the heading and body of procedures (and functions). The *begin* symbol has only one purpose, which cannot be modified by what follows it. In Algol 60, it is not obvious whether variables local to the body of a procedure are declared at the same level as parameters, or one level higher. Another source of difficulties is the fact that the introduction of a declaration at the beginning of a compound statement changes the scope of every label defined within this construct.

In fact, the resulting simplification and clarification in Pascal does not seem to be disadvantageous, since experience shows that for a program built in a modular and systematic way, the need for disjoint blocks which are not procedures seldom occurs. When it does, the price to pay is not heavy: define two procedures without parameters, one for each block, and call them in place of the blocks. The block then becomes a particular case of a more general construct. To define as procedures the modules used during program design is more general and natural than to replace them with disjoint blocks. In fact, it is a logical consequence of top-down design, which is one of the bases of structured and systematic programming.

3.2. *Dynamic arrays*

The bounds of an array declared in Pascal must be known at compilation time, so changing these bounds implies recompilation of the program. Although generations of programmers have submitted daily to this constraint when using Fortran, it is true that it is an inconvenience. Before condemning Pascal, however, it is worth examining the magnitude of this inconvenience, and to see whether the inconvenience is not compensated by some advantages.

First, we remark that it is the computer which must recompile the program, and not the programmer, and that moreover a Pascal compiler can be fast enough to compile and load a given program in a time comparable with the loading time required by a classic link-editor for an equivalent Fortran program (already compiled) [7,23]. Second, the possibility of constant and type declarations in Pascal makes it extremely easy to simultaneously change the bounds of one or several arrays, subrange declarations, limits of loops and any other points concerning the array; see for example the partial programs given in section 2.2 of this paper. Third, to be able to choose array bounds at run time frequently leads the programmer to leave the user with responsibility of choosing sufficient limits, checking that they are not overflowed, and even of counting his data by hand (computers count much better than people).

On the other hand , the knowledge of array bounds by the compiler itself allows more efficient access to individual elements. With variable bounds, access to an element necessitates a search in the allocation stack, at a place which is known only indirectly, for the value of each bound. For a two-dimensional array, this requires four memory accesses which are not required with fixed bounds. Another important point is to examine the consequences of allowing dynamic arrays in a language which has to be both general and simple: the implementation and behaviour of records or files with array components would become unbearably complicated, costly and error-prone.

3.3. *Conditional expressions*

Habermann judges that the statement *i* := *if* *i* = *7* *then* *1* *else* *i* + *1* expresses more clearly than the statement *if* *i* = *7* *then* *i* := *1* *else* *i* := *i* + *1* that a certain value is assigned to *i*. This point seems to be at the very least debatable, and our experience shows that in Algol 60 programmers use almost exclusively the second form, which has the advantage of allowing the replacement of either of the two assignments by a compound statement without modifying the other.

The only way to allow full generality in that sense would be (depending on the chosen point of view) to unify or to confuse the notions of statement and expression, yielding an expression language like Algol 68 [18] or Bliss [5]. The consequences of such a decision for the language structure and for its compilation are very complex, and exceed by far the doubtful advantage quoted before. As may be seen in several examples in [5] or [18], the normal use of an expression language yields formulas which are much too deeply parenthesized (be it explicitly or not) to be easily understandable. Just as the human mind can manage only small amounts of program at a time [13], it has a mental stack of a very limited depth. This is a more important reason for the lack of understandability of APL programs than the plethora of different operators.

3.4. *Labels and goto statement*

This precise point is the most characteristic of those aspects of programming languages about which reasonable people may disagree in all honesty. Excommunication of languages which include, or do not include, labels and *goto*'s is never the proper attitude to adopt. However, let us remark that Pascal restricts the use of labels severely. They are not at all manipulable objects, they must be declared in all cases [8,15], it is not allowed to enter a procedure or a structured statement from outside [15], and so on. Since you cannot prevent the users from writing bad programs if they like to do so, and since a *goto* exiting a procedure is the simplest way to handle error cases where the structure of the program must be irreparably broken [20], the choice to maintain labels and *goto*'s in Pascal is as well defensible as the other solutions currently proposed [5,24].

3.5. *Structured types*

As we said in section 2.3, the notion of type in Pascal includes three different concepts, which we call type, range and structure, to use the same terminology as Habermann. The decision to name "type" what is in some cases a description of a storage template, namely for all structured objects (arrays, records, sets and files), is indeed debatable, but the important question is not the appropriateness of the particular labels chosen. More important is the fact that a structured object can, in all sensible situations, be handled as an unstructured one. For example, an array may be a component of a file or another array, or a field of a record; assignments are valid for almost all objects, provided that the left-hand and right-hand sides have the same type, consequently one can in a single statement copy a record (without variant) or an array; some operators may have structured operands, for example = and ≠ for records, or all the comparison operators for packed arrays of characters (called "strings" in many languages).

The result of the approach chosen in Pascal is that the whole notion of a data type is simple and coherent, as may be seen in the tree diagram at the end of section 2.3. In this schema, all structured types are made from other types, which can themselves often be structured, but ultimately lead to unstructured types, and from there to either primitive types of enumeration-defined types. The set of primitive types could indeed be extended to include, for example, complex numbers, but they have no counterpart in most hardware, and may be simulated at a very small cost with the data structuring tools offered by Pascal. We do not think that the lack of distinction between types, ranges and structures, by labelling all of them as type, is a source of confusion, but it may hinder somewhat the understanding and explanation of subrange type particularities.

3.6. Side-effect in functions

While the original Report on Pascal [6] recommended that a function make no modification to non-local variables (but did not pretend that they could not), the Revised Report does not say anything on the matter. In fact, to enforce such a restriction would be extremely difficult and costly, if not impossible, unless one forbids variable parameters and procedure calls within the body of a function. This is another instance of the situation where you cannot prevent the user from writing silly programs, unless you prevent him from writing any program at all. Moreover, the Revised Report allows declaration and call of a parameterless function, which is of no use if it cannot modify any non-local variable. Such functions are useful in some cases, and several standard functions have (or sometimes have) no parameter.

4. Valid criticisms

This section deals with points which are indeed deficiencies in Pascal, and should perhaps be changed in a future version of the language (if possible). The brevity of this section is in itself a good argument in favour of Pascal.

4.1. Array parameters

Since the bounds of an array are part of its type (or, more exactly, of the type of its indexes), it is impossible to define a procedure or function which applies to arrays with differing bounds. During two years of intensive programming in Pascal, we encountered only one case where this restriction was of any importance. The reason is probably that, because array bounds are static, different arrays with components of the same type generally have the same bounds, not exactly fitted to the set of data during a precise run. To suppress this limitation would probably be extremely difficult, and not worth the trouble.

4.2. Variable initialization

Pascal does not presently allow initialization of variables at compilation time, at least in its official version. The richness of data structuring tools makes such a possibility very difficult to define, but it is indeed in the course of study, and will probably be available before long [15].

4.3. Parametric procedures

Pascal presently contains, in one respect, a lack of rigorous specification which either leads to a certain inefficiency, if one wants to do all the necessary checks, or to a certain insecurity if they are not all done. In the specification of a function or procedure passed as a parameter (we shall say "parametric procedure"), the type and number of parameters are not specified at all, so it is generally impossible to easily detect at compile time the following error (this example is ours):

```
procedure P (procedure Q); begin Q(2,"A") end;
procedure R (x:  Boolean); begin write (x) end;
   begin P(R) end.
```

Some restrictions have already been made to the use of parametric procedures : a parametric procedure was at first not allowed to have procedure or function paramaters [6], and now it cannot have variable parameters either [15], leaving only value parameters. While useful and not constricting, these restrictions do not suffice to ensure complete security, and they are not made explicit in the syntax. However, it is very easy to make the simple syntactic modification which appears below, redefining the non-terminal < *formal parameter section* > in section 10 of the Revised Report.

```
< formal parameter section > ::=
   < parameter group > |
        var < parameter group > |
        function < procedure skeleton > :
                  < type identifier >
              {, < procedure skeleton > :
                  < type identifier > } |
        procedure< procedure skeleton >
              { ,< procedure skeleton > }
< procedure skeleton > ::= < identifier > |
        < identifier > ( < type identifier >
              { , < type identifier > } )
```

With this modification, the restrictions quoted before appear explicitly in the syntax, and the heading of our procedure *P* must be either

```
   procedure  P(procedure  Q(integer,char))
```

which will allow detection of an error when *P* is called with *R* as a parameter, or

```
   procedure P(procedure  Q(Boolean))
```

which will allow rejection of the call to *Q* in the body of *P*. The verification of compatibility between formal and actual parameters of parametric procedures can thus be made completely (and cheaply) at compile time, even with a one-pass compiler, if it adopts the convention of pre-declaring procedures (see [2], section 13).

5. Misunderstandings and minor errors

We shall consider in this section only the misunderstandings and errors made by Habermann which may lead the reader to a false idea of Pascal. We shall ignore many petty points, and make no comment about the general philosophy of the paper itself.

5.1. Syntactic errors in examples

In Pascal, all declarations precede the body of a procedure or of the program. Thus, in both examples of section 3, *begin* should occur after the declarations. Similarly, a *begin* should be placed in front of the last line of the example in section 6. In section 2.2, however, this is done correctly.

One error is repeated consistently throughout the whole paper: the lower and upper limits of a subrange (in a type declaration or as an index type) should be separated by ".." instead of "...".

In fact, when it is said in section 3 that a program "results in an error indication at the operator +", an actual Pascal compiler would have indicated four errors before encountering this operator (*begin* before declarations; "..." in a subrange twice; and $i = i + 1$ instead of $i := i + 1$), and none at that point.

5.2. Errors concerning the notion of type

At the beginning of section 5, it is said that, since scalar types are subranges, the declaration

var A : array [real] of integer

is legal. Of course, this is false: the type *real* constitutes a singular case of scalar type, since "the number of values of this type is unknown, and there is no unique ordering among these values" [15].

Similarly, it is said near the end of the same section that, since a simple type may be represented by a type identifier, a file or array type may serve as index for an array. This is patently absurd, and it is evident that a type identifier does not always represent a simple type.

In section 7, it is said that "a constant has no type". This is obviously false, and the Report clearly specifies in section 4 the type of the value represented by each kind of constant.

5.3. Miscellaneous errors

In the middle of section 4, a question is asked "wheter or not a label in front of the statement part of a procedure declaration is considered as in the procedure or not". The answer is that a label is forbidden in such a place, as the Report clearly states.

In section 7, it is asked why the symbols * and ⊕ are introduced, since the notation { } was introduced just before. The answer is that the braces apply only to a sequence of symbols, while * and ⊕ apply to one symbol only.

6. Conclusion

The main point to note in conclusion of this reply is that the Report on Pascal is aimed to serve both as a defining document and as a manual and tutorial for programmers. Such a paper must necessarily rely on some natural good will on the part of the reader, unless it is to grow into PL/I-like dimensions [9,21] or Algol 68 unreadability [18].

The second point is that the Report has an indispensable complement and companion, the Axiomatic Description [8], which defines in a rigorous manner all the semantics of Pascal and occupies only nine printed pages. Moreover, it is quite readable.

Acknowledgments. Our colleague Neil Stewart was most helpful in improving the style and correctness of this paper. Jim Horning was even more helpful by strongly criticizing the emotional and technical contents of a first version.

References

1. Habermann, A.N.: Critical comments on the programming language Pascal. Acta Informatica 3, 47-57 (1973).
2. Wirth, N.: The programming language Pascal (Revised report). Berichte der Fachgruppe Computer-Wissenschaften, Eidgenössische Technische Hochschule, Zürich (December 1973).
3. Habermann, A.N.: Prevention of system deadlocks. Communications of the A.C.M. 12, 7 (July 1969).
4. Habermann, A.N.: Synchronisation of communicating processes. Communications of the A.C.M. 15, 3 (March 1972).
5. Wulf, W.A., D.B. Russell and A.N.Habermann: Bliss: a language for systems programming. Communications of the A.C.M. 14, 12 (December 1971).
6. Wirth, N.: The programming language Pascal. Acta Informatica 1, 35-63 (1971).
7. Wirth, N.: The design of a Pascal compiler. Software practice and experience 1, 4 (October 1971).
8. Hoare, C.A.R. and N. Wirth: An axiomatic definition of the programming language Pascal. Acta Informatica 2, (1973).
9. PL/I language specifications. IBM Corporation, Form C28-6571.
10. Dijkstra, E.W.: The humble programmer. Communications of the A.C.M. 15, 10 (October 1972).
11. Holt, R.C.: Teaching the fatal disease (or) introductory computer programming using PL/I. Sigplan Notices 8, 5 (May 1973).
12. Wirth, N.: From programming techniques to programming methods. International Computing symposium 1973, edited by Günther et al., North-Holland, Amsterdam (1974).
13. Dijkstra, E.W.: Notes on structured programming. Structured programming, by Dahl et al., Academic Press, London (1972).
14. Hoare C.A.R.: Proof of a structured programming : the sieve of Eratosthenes. Computer Journal 15, (1973).
15. Jensen, K. and N. Wirth: A user manual for Pascal. Institut für Informatik, Eidgenössische Technische Hochschule, Zürich (April 1974).
16. Wirth, N.: Systematic programming - An introduction. Prentice-Hall, Englewood Cliffs (1973).
17. Dijkstra, E.W.: A short introduction to the art of programming. Department of Mathematics EWD-316, Technische Hogeschool, Eindhoven (1971).
18. van Wijngaarden, A. et al.: Report on the algorithmic language Algol 68. Numerische Mathematik 14, 1 (February 1969).

19. Hoare, C.A.R.: Notes on data structuring. Same reference as [13].
20. Hoare, C.A.R.: Recursive data structures. Computer Science Department CS-400, Stanford University (October 1973).
21. PL/I(F) reference Manual. IBM Corporation, Form C28-8201.
22. Naur, P. (editor): Revised report on the algorithmic language Algol 60. Communications of the A.C.M. 6, 1 (January 1963).
23. Hoare, C.A.R.: Hints on programming language design. Computer Science Department CS-40, Stanford University (December 1973).
24. Leavenworth, B. (editor): Control structures in programming languages. Sigplan Notices 7, 11 (November 1973).