EUROPEAN ORGANISATION FOR NUCLEAR RESEARCH

CO/Note 80-34 /Rev.
23.1.1981

How to avoid getting SCHLONKED by PASCAL

R. Cailliau

Abstract

The programming language Pascal and its derivatives are "IN": we are in a phase of "wild enthusiasm". However, in many fields of programming, Pascal presents problems, which may easily offset its advantages. Users are warned of lurking dangers and impossible to overcome barriers. Programming domains where Pascal is nevertheless outstanding are suggested.

(text formatting by REPORT program)

Contents

1. Introduction 1

2. Explanation of the title 1

3. History of Pascal 3

4. The popularity of Pascal 4

5. Some good features 5

6. Design aims 10

7. The problems 12

7.1. Definition of the language 12

7.2. Problems on the coding level 15
7.2.1. The unclosed comment: 15
7.2.2. The forgotten mechanism specifier: 15
7.2.3. The range boundaries violation 16
7.2.4. The automatic importation feature 17

7.3. Problems with types 19
7.3.1. Operations 19
7.3.2. Structures 20

7.4. Missing useful constructs 23

7.5. Some miscellaneous points 26

7.6. Problems in managing the coding 30
7.6.1. Compile time expressions 30
7.6.2. Library functions 31
7.6.3. Separate compilation 31

8. Useful application areas and conclusion 34

9. References 36

1. Introduction

This paper was the basis of two talks, one a seminar at the Data and Documentation (DD) division of CERN, the other an invited lecture at the meeting of the Nord Computer Users Society (NOCUS). The first talk was held in May 1980, the second in October 1980. Some material has been added, notably some examples which are difficult to present during a one-hour presentation.

The format of the oral presentation explains the use of the first person singular and the presence of some figures which were only intended to keep the audience awake...

Furthermore, this text does not make a claim to completeness, it treats only some aspects, viz. those which are easily communicated. Some deeper problems are not touched. No aspects of theoretical computer science are dealt with, the emphasis is on the human engineering side.

2. Explanation of the title

On one of our computers there is a game-program called the TWONKY. In this game, you are supposed to escape from a maze. In the maze lives the TWONKY, a monster, which wants to "absorb" you. You cannot see the TWONKY and you must try to escape before the TWONKY gets you. The TWONKY can move through the walls of the maze and is not hindered by a number of obstacles.

When the TWONKY gets close enough, it will gobble you up in one bite, and you have then been SCHLONKED.

I feel that today there are quite a number of programmers who tend to think that Pascal can actually help you escape from the maze of programming. It may be, however, when you rely too much on the powers of Pascal, that you run into unforeseen difficulties and in the end you may get SCHLONKED.

I do not want to paint a picture of Pascal as a monster programming language which should be avoided at all costs (and here the TWONKY analogy ends, fig. 1) but it seems only fair to put a damper on the unlimited enthousiasm with which Pascal is sold today.

Fig. 1

3. History of Pascal

Pascal is a computer programming language in the Algol-style. It was developed by Prof. N. Wirth as the result of a long experience in language design and compiler writing. I quote from [1] (Wirth, 69, page 455):

> The language Pascal is the latest product of a research and development project that was initiated about eight years ago and led to the languages EULER, ALGOLW, PL360 and others that never reached the state of publication and wider use.

Here is a short overview of Pascal´s development:

- the first version was drafted in 1968,
- the first paper written in 1969, published in 1970,
- the first compiler became operational in 1969.
- a revised report was published in 1973, together with an axiomatic definition.
- a User manual & Report were printed in 1974.
- an ISO standard was drafted in 1977. It was released for public comments in 1979, reworked and proposed in 1980 (fifth version).

Pascal´s place among other languages is shown in fig.2.

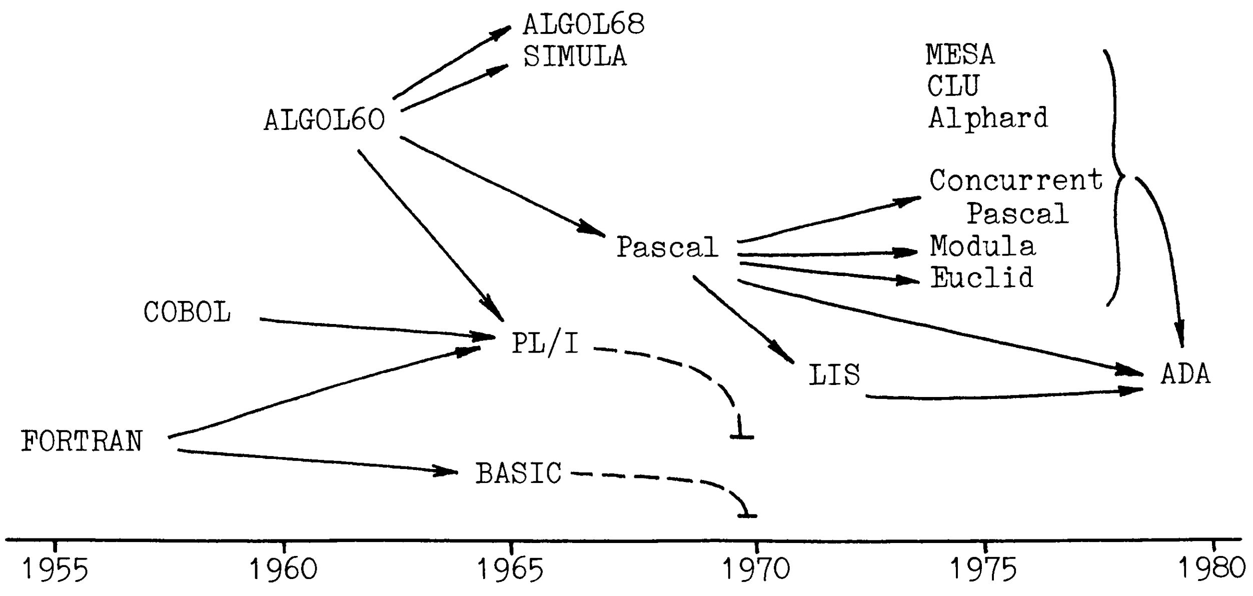


Fig. 2

Fig. 2 represents only the mainstream of "algorithmic" languages, that is why APL, LISP etc. are not included.

Since Pascal´s success, we have seen attempts at new languages, trying to capture the essence of Pascal or trying to apply it to other domains: Euclid and Modula among others, and now ADA.

4. The popularity of Pascal

D. Bates and I implemented Pascal on the Nord computers of the PS division of CERN in 1976 [8] and we have since had 4 years of experience with it.

The success of Pascal can be attributed largely to the following factors:

- small compiler, hence useable on minis and micros,
- portable compiler, easily bootstrapped,
- "no" costs for portable compiler,
- existing literature (User manual & Report),
- small language (can be remembered in its entirety),
- compiler is written in Pascal and is readable,
- used at many universities for teaching,
- it is a good language !

5. Some good features

Pascal does have some excellent features. Here is a list of positive aspects :

- small number of well-chosen keywords,
- small number of syntax & semantics rules, very few exceptions to these rules (orthogonal),
- meaning of Pascal instructions is highly independent of environment, which promotes protability of programs,
- excellent data structuring methods,
- clean and efficient control structuring,
- excellent for programming "in the small",
- gives a feeling of reliability,
- with some care, readability can be kept high.

For in fact, when one studies the activity of program development, then one can perceive the task of obtaining the program source text as the result of a translation effort (fig. 3).

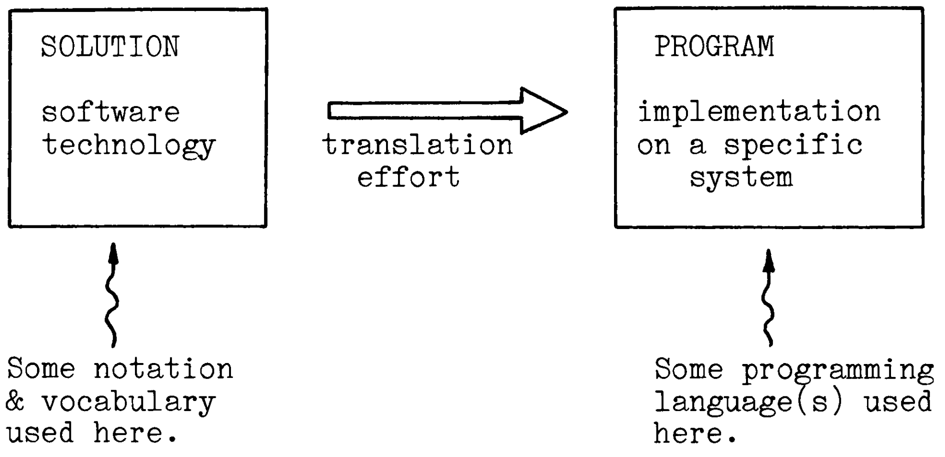

SOLUTION
software technology
translation effort
PROGRAM
implementation on a specific system
Some notation & vocabulary used here.
Some programming language(s) used here.

Fig. 3

First a general solution is found to solve the problem that was given. This solution satisfies the boundary conditions of current software (and hardware) technologies. It should be independent of the features of any programming language. The solution is written in some natural language together with some notation which makes it hard to read for the layman: it is already specialized (sometimes even already machine-dependent!). The program to be obtained must be a much more detailed form of the solution and is written using one (or more) programming languages.

Good practice requires that the solution be so detailed and has been worked out so well, that the program reflects the design of the solution point by point.

Clearly then the program is some translation of the formalized solution. The translation effort will largely depend on the similarity of the jargon used in the statement of the solution and the programming language.

Let us give an example:

Suppose we want to treat information about a hundred people, and that we want to represent for each person his or her name, age and sex. Then a data item for one person looks structurally like:

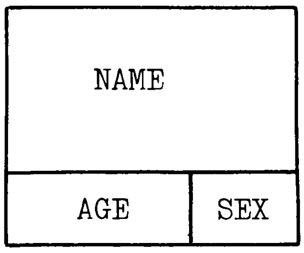


Fig. 4

Since we want to do this for a hundred people, we need an array, which we could picture like:

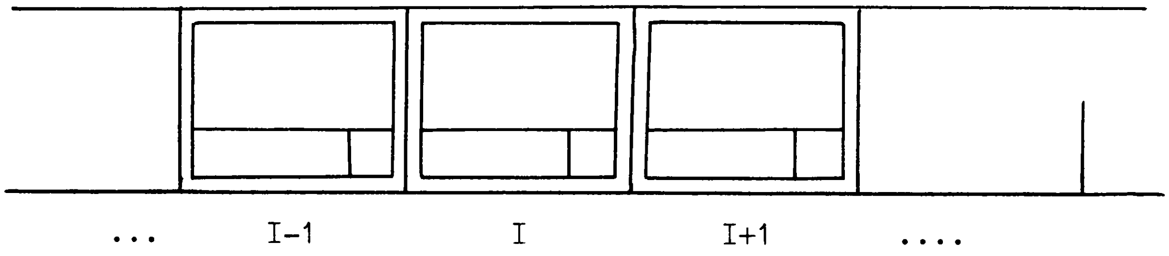


Fig. 5

In Fortran this could be done by the following bits of program:

```
CHARACTER*20 NAME(100)
INTEGER AGE(100)
LOGICAL SEX(100)

...
MALE = .TRUE.
FEMALE = .FALSE.
...

...
NAME(I) = 'PASCAL'
AGE(I) = 11
SEX(I) = MALE
...
```

Fig. 6

In Pascal we would use:

```
TYPE PERSON = RECORD
              NAME: ARRAY [1..20] OF CHAR;
              AGE:  0..150;
              SEX:  (MALE, FEMALE)
              END;

VAR PERSONLIST: ARRAY [1..100] OF PERSON;

...
WITH PERSONLIST[I] DO
   BEGIN
   NAME := 'PASCAL              ;
   AGE := 11;
   SEX := MALE
   END;
...
```

Fig. 7

Note that if we wanted to build a list or a file or a tree of people-items, then we do not have to modify the Pascal record for PERSON. In Fortran we would have trouble. In fact, we already have a translation problem in the Fortran example, because we must express the array of a collection as a collection of arrays, and accesses to the data of one person are definitely less efficient: the array indexation has to be repeated three times, whereas it is done only once in Pascal. (If one wants to achieve the efficiency, then the Fortran compiler must do such optimisations as the recognition of common index expressions in arrays of different element sizes).

Another example is that of range checking. In the handling of arrays, the index is not supposed to exceed the array index bounds.

To ensure this condition one can make a check at each indexation into the array, or one could, more efficiently, specify that the index variable must stay within range. In the latter case a check need be made only at the assignments to the index variable. This can be achieved easily in Pascal by using a range in the declaration of both the arrays and the corresponding index variables:

```
TYPE INDEX = 1..10;

VAR X,Y,Z: ARRAY [INDEX] OF T;
    I: INDEX;

...
I := J*2+K;           <-- check done here,
...
X[I] := Y[I] + Z[I];  <-- NO checking necessary here
...
I := 20;              <-- can even lead to compile-time
                          error message "out of range".
...
```

Fig. 8

The ability to express the constraints on certain important variables definitely increases readability and reliability.

6. Design aims

If one reads an article on Pascal one is bound to find somewhere a statement about Prof. Wirth´s aims when he designed the language. Usually, the author will mention those purposes of Pascal which best justify the writing of the article. This would seem quite normal, and there is nothing to be surprised about in the fact that different authors list different sets of design goals. What is more disconcerting is that successive articles of the <u>same</u> author present us with different lists.
Let me quote from some of the best known of Prof. Wirth´s own articles:

October 1969 [1]:

- Clarity and rigour of description;
- range of applicability: it should be possible to express in Pascal anything that can be written in machine code;
- efficiency: few features require run-time software routines; no type information needs to be held at run-time;
- processor reliability (compiler);
- machine independence;

October 1970 [2]:

- to make available a language suitable to teach programming as a systematic discipline (sic);
- to develop implementations which are reliable and efficient on present day computers;

July 1971 [3]:

- to make available a notation in which fundamental concepts and structures of programming can be expressed;
- to make available a notation which takes into account the various new insights concerning program development;
- to demonstrate that a language with a rich set of flexible data and program-structuring facilities can be implemented by an efficient and small compiler;
- to demonstrate that a compiler written in such a language is more readable, efficient and reliable;

- to gain more insight into the methods of organizing large programs and managing software projects;
- to obtain a home-made tool which can (due to its modularity) be adapted to various needs (the compiler).

July 1973 [5]:

-same as 1970.

June 1975 [7]:

- to promote the writing of programs with languages that facilitate transparent formulation and automatic consistency checks.

However, (fig. 9) we shall see in the remainder of this talk how well any of these sets of aims are satisfied.

Fig. 9

7. The problems

The examples I will show in this section are by no means the most common ones or the ones most difficult to detect, they are the ones most easily communicated.

7.1. Definition of the language

The Revised Report of 1973 is very different from the first report, which is acceptable since it is marked as revised. However, it is quite remarkable that :

1. the User Manual and Report describes yet another language (in the details);
2. the portable compiler which was distributed from Zurich both after the Revised Report [5] and the Manual & Report [13] compiles a fourth language.
3. the ISO standard has just gone through its fifth draft [11], [12]. I read thouroughly the fourth draft, and I find the fifth to describe a language which is again different from that of the fourth.

Furthermore, the evolution of the length of the documents describing Pascal shows alarming trends:

- the original report has 28 pages,
- the report part of the Manual & Report [13] has 32 pages,
- the fourth ISO draft standard has 44 pages, this sudden increase was necessary to remove implementor´s doubts about what to do in certain cases left unspecified in the original reports,
- the fifth ISO draft standard has 66 pages, because the fourth one was vague in many areas. The fifth draft still leaves a large number of decisions to the implementor.

The following figure shows the trends more dramatically, and I think everyone will know which function best fits this curve:

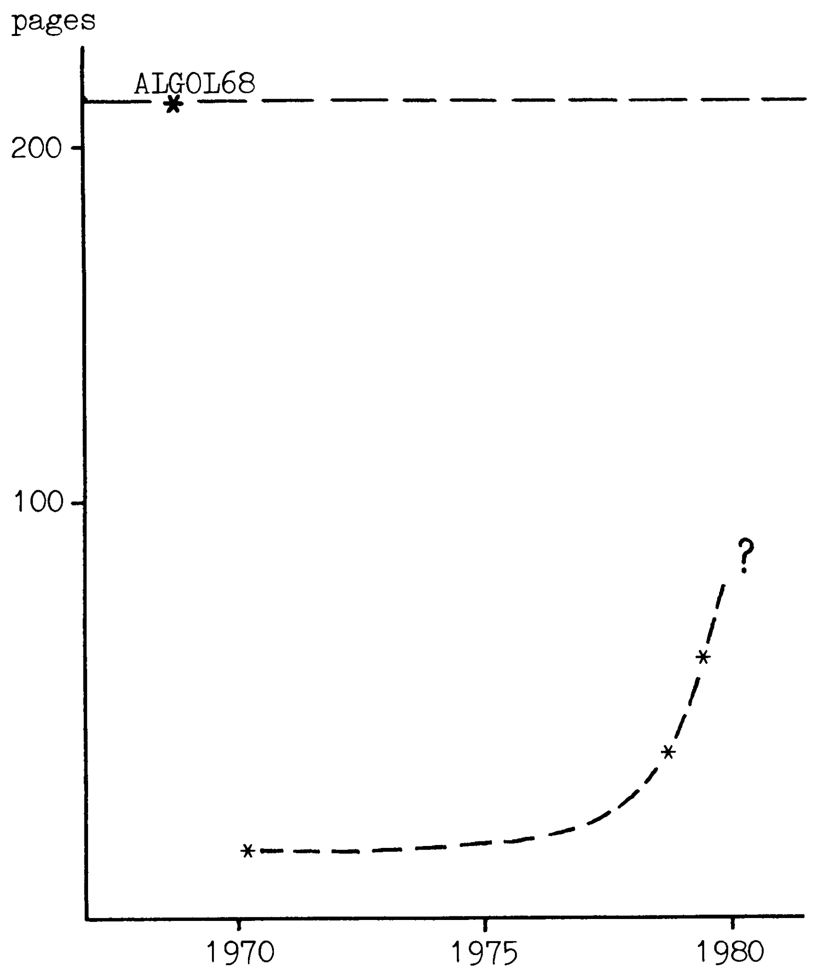


Fig. 10

ALGOL68 was described in great detail in its first report, even using a tailor-made jargon in an attempt to remove all ambiguities. It has been said that ALGOL68 was an "impossible" language because of the "obscurity" and length of its defining document. The last draft of the Pascal standard is at least as "obscure": here is a typical sentence (though taken out of context):

> The occurrence of an identifier as part of the identifier-list of a variable-declaration shall be its defining-point as a variable-identifier of the given type for the region that is the block immediately containing the variable-declaration-part in which the variable-declaration occurs.

It may be that any language needs a thick report if one wants to define it properly. Unfortunately, ALGOL68 started off by the publication of that document, and that may have been very bad publicity.

Pascal started with an almost simple-minded definition, omitting the discussion of many "hairy" cases of Pascal programming. But at least the document showed the usefulness of the language.

These hairy cases unfortunately are now SCHLONKING the ISO-standard writers. They are also the subject of this talk.

7.2. Problems on the coding level

All examples shown here present problems which occurred more than once in our environment at PS division.

7.2.1. The unclosed comment:

The conventions for writing comments are not agreed upon, there exist at least four different sets of rules. In this example the "standard" is assumed to hold:

```
...
I := J - K ;            (* REMEMBER DIFFERENCE
J := K ;                (* RESET J *)          ^
...                                            |
                                               |
                                        comment not
                                        closed here
```

Fig. 11

The comment on the first line is closed on the second, and the statement J:=K is "commented away" accidentally. Of course, by Murphy´s law, this is the kind of error that goes undetected through all tests and that then explodes the program on a Saturday at midnight...

7.2.2. The forgotten mechanism specifier:

Apart from the comments one could make about the unfortunate choice of the parameter passing conventions, there is a danger in the simple rule that a default will be used when no mechanism is specified. Consider fig. 12:

```
PROCEDURE INCREMENT (VAR V: INTEGER);
   BEGIN
   V:=V+1
   END;

...
INCREMENT(LINECOUNT);
...
```

Fig. 12

In this example, the procedure increments the value of its parameter by one. If the VAR is present, then the actual parameter will be affected, otherwise a copy will be made and only the copy will be affected! Therefore it is not enough to read the line

V:=V+1

to ascertain that LINECOUNT has indeed changed: inspection of the list of parameters is of paramount importance. In large programs this bug leads to strange results, and it is hard to detect.

7.2.3. The range boundaries violation

The programmer who has declared range limits for his variables feels safe: Pascal will check that the value will not transgress the limits. But how many compilers are clever enough to catch the following violation:

```
VAR I: 1..10;
...

PROCEDURE P(VAR Z: INTEGER);
   BEGIN
   Z:=100
   END;

...
I:=5;
WRITE(I);            <-- will output 5,
P(I);
WRITE(I);            <-- will output 100..?
...
```

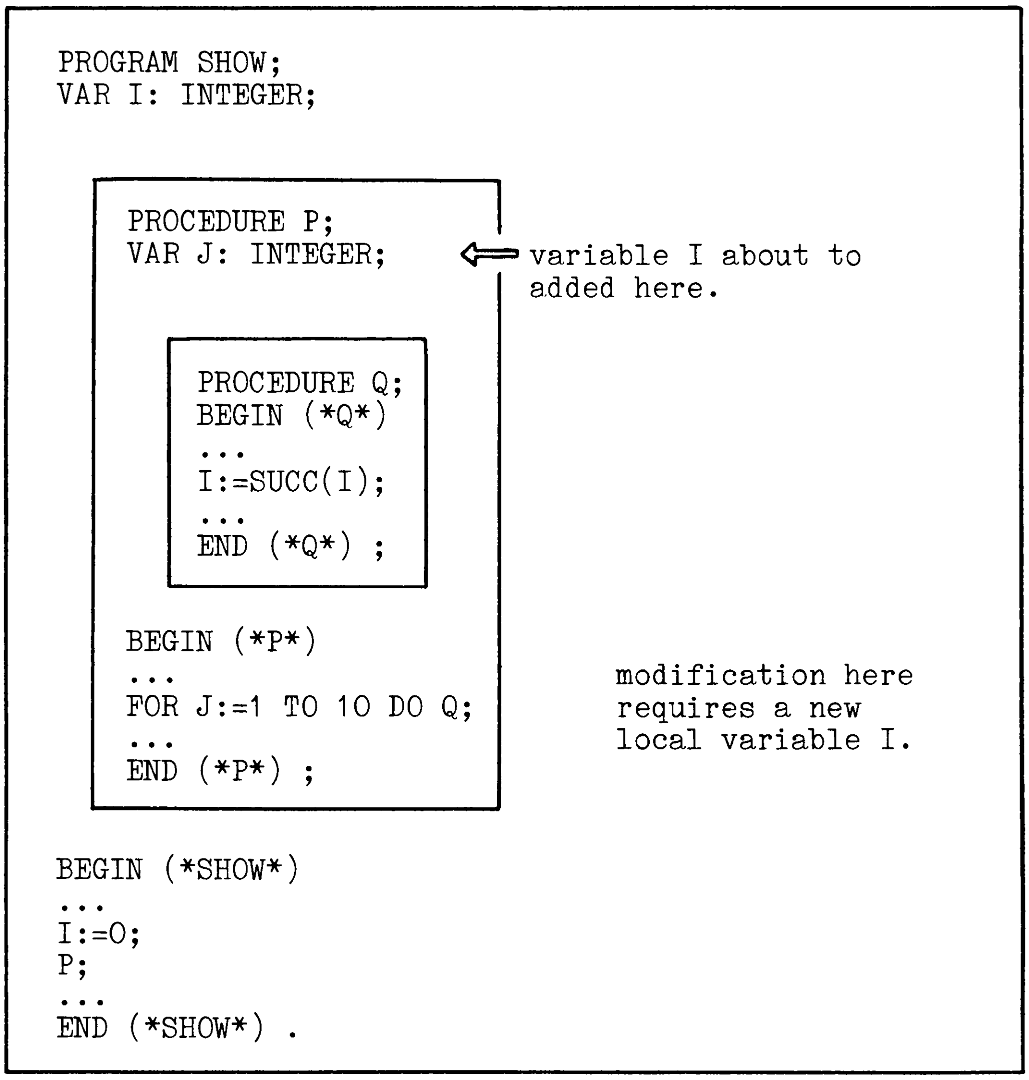
Fig. 13

7.2.4. The automatic importation feature

The scope rules of Pascal are more or less taken from Algol 60. This implies that:

```
procedures may declare local variables,
           may be nested statically,
           may access variables declared at outer levels.
```

In Fig. 14 sections of a program are shown:



```
PROGRAM SHOW;
VAR I: INTEGER;

  PROCEDURE P;
  VAR J: INTEGER;

    PROCEDURE Q;
    BEGIN (*Q*)
    ...
    I:=SUCC(I);
    ...
    END (*Q*) ;

  BEGIN (*P*)
  ...
  FOR J:=1 TO 10 DO Q;
  ...
  END (*P*) ;

BEGIN (*SHOW*)
...
I:=0;
P;
...
END (*SHOW*) .
```

Fig. 14

Procedure Q sits inside P but does something with the global variable I. Suppose we know Q to be correct. Now one day a modification to P is necessary, for which we read the body of P only: if the program has been constructed in a decent way, we should be able to limit the modification to the body of P, i.e. we do not have to inspect the code outside P, nor should we have to look at the code (of Q) nested inside P.

Furthermore, suppose the modification needs the declaration of a new integer, local to P, and that we call it I. Under the current standard, it has to be declared *before* the declaration of Q. That will make I visible to Q, which is not necessary. Much worse, by calling it I, Q will now no longer see the global I but the I local to P instead. The modification may be entirely correct but the program may (or it may not!) blow up inexplicably. The compiler cannot object... A possible solution would be to require an explicit declaration of the importation of a global or intermediate variable, so that by reading the declarations local to P we would see that it already imports I for use by Q.

7.3. Problems with types

7.3.1. Operations

Here is an example which appears in many introductory texts on Pascal, and which I would at least call "malicious advertising":

```
TYPE COMPLEX = RECORD
               REALPART, IMAGPART: REAL
               END;

VAR A,B,C: COMPLEX;


A := B*C;              <-- not possible

...
MULTIPLY(B,C,A);       <-- possible, but not
...                        convincing...
```

Fig. 15

It is possible to define a type COMPLEX (Pascal has no built-in type COMPLEX!) and to declare variables of that type. The snag is in the line of the example which shows the "*" operator. This line does not appear in any of the textbooks, because it is not permitted.

Clearly it is not sufficient to be able to define new types, one must also be able to define the operators on these types. Most of the operators will be expressed by procedures (subroutines), but the algebraic operator notation (infix) is very useful for types such as complex, matrix, list structures, etc.

7.3.2. Structures

One of the much publicized features of Pascal is its ability to deal with sets. I have found Pascal sets to be extremely useful for all kinds of programs. Consider the following:

```
VAR LETTERS: SET OF CHAR;
    CH: CHAR;

...
LETTERS := ['A'..'Z', 'a'..'z'];
...
IF CH IN LETTERS THEN ...
...
WHILE CH IN (LETTERS + ['0'..'9']) DO ...
...
```

Fig. 16

In fig. 16, the variable LETTERS can contain any subset of the implementation defined set of characters. In the third line of fig. 16, it is assigned the set consisting of the upper and lower case letters.

(Note: Implementation of the SET structure is usually done by a string of bits, whereby the elements in the set are represented by bits that are "on", the other elements have their corresponding bits "off". Thus a SET OF CHAR would (for ASCII) always have 128 bits. After the assignment to the variable LETTERS, its bits 65 to 90 and 97 to 122 would be "on", the others "off". On a 16-bit machine, such a variable would take up 128/16=8 words.

In the IF-statement of fig. 16, if the variable CH contained the character 'X', which is represented by ASCII code 88, a test would be made to see if the 88th bit of LETTERS was "on" or "off". This is fairly straightforward to implement, and fast. The set operations of union, intersection and so on all can be realized efficiently by the normal bit operations on machine words (end of note)).

But the SET structure deals only with the mathematical, <u>static</u> sets. These are sets of identifiers (values) <u>not</u> sets of <u>objects.</u> The set of objects, which is a useful entity to <u>have</u>, is absent from Pascal (but not, for example, from PLANC [16]). Thus it is impossible to create a number of objects such that they have attributes which may vary in time, and then to construct a set of some of them.

Of course, one can add a Boolean to the description of each object, to indicate set membership. But this does not permit operations on the whole set. The only solution here would be to add a pointer and construct the set as a linked list of all members. But then one is not allowed to point to named variables... Thus a program containing the following fragments is not possible:

```
TYPE INTERFACEBOARD = RECORD
                      MANUFACTURER: ...
                      SERIALNUMBER: ...
                      LASTDATEREPAIRED: ...
                      ...
                      END;

VAR MAINTENANCE: FILE OF INTERFACEBOARD;
    M: INTERFACEBOARD;
    BROKENBOARDS: SET OF INTERFACEBOARD;  <-- impossible.

...
BROKENBOARDS := BROKENBOARDS + [M];
...
IF M IN BROKENBOARDS THEN ...
...
```

Fig. 17

The translation effort is here notably higher than for an intermediate level language (such as PL-11 or Nord-PL).

It is impossible to solve the above problem by defining a new structure such as VARSET, or, say, LIST. If one could do that, then declarations like the following would be possible:

```
VAR BROKENBOARDS: LIST OF INTERFACEBOARD;
```

in this, LIST would be a structure such as ARRAY or FILE. It must be said that so far, no language has solved the problem of defining new structures satisfactorily. This is mainly due to the problem of inventing a syntax for the user-definition of such structures and the problem of supplying the information for its implementation in a more or less machine independent way.

As I said, sets are extremely useful and one gets easily addicted to them. There are however other problems with sets that lead to frustration: the range or the upper and lower bounds of values that may be members are usually rather limited. The whole set construct is one of the least portable in the language.

Some implementations forbid even SET OF CHAR and that definitely makes any program using SET OF CHAR unportable. Furthermore, it is quite hard to program around the limitations. The ND Pascal implementation [10] is one of the few where the user can tell the compiler how big he wants his sets. Of course, it is the job of the compiler to find out how big sets are. Unfortunately, it cannot do so because of displays of sets of integers, such as:

```
[ 1, 2, 100, I*J ]
```

It is impossible to find out whether this now is a value for a set of type

```
TYPE SMALLSET = SET OF 1..100;
```

or whether it is of type

```
TYPE LARGESET = SET OF -1000 .. +1000;
```

this problem can only be solved effectively by requiring the type identifier to be specified together with the construct, as in:

```
SMALLSET([ 1, 2, 100, I*J ])
```

or

```
LARGESET([ 1, 2, 100, I*J ])
```

which would leave no doubt, and is also more readable. Let us note that ADA still does not adopt this attitude towards structured constants, and therefore it already has problems in identifying overloaded operators [14].

7.4. Missing useful constructs

This is a dangerous section, since it is always easy to come up with a large number of "features" that are "missing". They usually turn out to be little additions catering to specific needs of the person proposing them. I have myself thought of many such additions, sometimes I have introduced them, sometimes not. However, the following two I am prepared to defend on the grounds that I believe that they are generally useful: the inverse of ORD and the LOOP construct.

The inverse of ORD: an interesting asymmetry exists in the functions on enumeration types. One of the great advances of Pascal is the introduction of the user-defined enumeration type such as DAY shown here:

```
TYPE DAY = (Monday, Tuesday, Wednesday, Thursday,
            Friday, Saturday, Sunday);

(* Note that:
   ORD(Monday) = 0,  ORD(Tuesday) = 1, etc.
   SUCC(Wednesday) = Thursday  ("next" day of Wednesday)  *)

VAR TODAY: DAY;
...
FOR TODAY := Monday TO Friday DO ...
...
MENU[Friday] := ...
```

Fig. 18

Now enumeration types are nothing more than a renaming of the integers, but the increase in program readability and reliability that may be obtained by using them is quite dramatic. I would say that no other single feature of Pascal contributes more to the "Pascal flavour" of a language.

There exists a number of functions on enumeration types, among which ORD (ordinal value) which generates no code whatsoever. It is only made necessary by the typing mechanism to obtain the integer representation of an enumeration value.

However, the inverse of ORD does not exist, except for the type CHAR. (Probably there it was introduced because the compiler could not have been written without it!). The absence of this inverse is a nuisance in certain practical cases.

Consider:

```
TYPE INDEXTYPE = MIN .. MAX;
VAR VECTOR: ARRAY [INDEXTYPE] OF ITEMTYPE;
    ITEM: ITEMTYPE;
    LEFT, RIGHT, MEAN: IDEXTYPE;

...
LEFT := MIN;  RIGHT := MAX;
REPEAT
      MEAN := (LEFT + RIGHT) DIV 2;
      IF ITEM < VECTOR[MEAN]
         THEN RIGHT := MEAN
         ELSE LEFT  := SUCC(MEAN)
UNTIL ITEM = VECTOR[MEAN]
```

Fig. 19

The simple binary search in the array VECTOR can only be performed if the index is of type INTEGER, because the mean value cannot be found for any other enumeration type (except by introducing grossly inefficient and unreadable code). Thus it would be impossible to perform such a search if VECTOR had used the type DAY as index. The problem can easily be solved by creating automatically with every enumeration type declaration the corresponding inverse function for ORD. For fig. 19, the function DAY would be introduced by the compiler, whereby

DAY(0) = Monday, DAY(1) = Tuesday, etc.

so that in general:

for any enumeration type T defined by

T = (v0, v1, v2, v3, ... , vn);

we will have:

ORD(vi) = i; (*existing function ORD*)

T(i) = vi;   (*nonexisting inverse of ORD*)

note that no code need be generated for the T-functions.

The LOOP construct: Pascal has three looping statements, REPEAT, WHILE and FOR. The WHILE is claimed to be the simplest (it can be argued in fact that REPEAT is simpler) and also the one which corresponds to the structured programming precepts. But the most fundamental loop of all, the one which does not end implicitly, is missing. Thus, when one writes process control programs, many of which contain indefinite loops, one is forced to use weird constructs such as

```
WHILE TRUE DO ...
```

the other obvious problem not solved by existing constructs is that of the one-and-a-half cycle: deciding to stop a loop somewhere half-way through.

As it is, Pascal offers only two solutions:

- introducing flags and tests
- constructing LOOP by using GOTO.

Neither of these solutions is attractive nor reliable. The first one is in addition also inefficient.

The construct introduced in the PS implementation is very simple:

```
LOOP construct:                 equivalent construct
                                using GOTO:


LOOP <name> :                   111:

   S1;                             S1;
   ...                             ...
   ... EXIT <name>;                ... GOTO 222;
   ...                             ...

   S2;                             S2;

   ENDLOOP;                        GOTO 111; 222:
...                             ...
```

Fig. 20

The semantics are simple. The compiler can easily check that no EXIT statement occurs outside the defining loop, and because loops have names, it is possible to indicate exits from more than one level.

7.5. Some miscellaneous points

Boolean operators: one of the more irritating holes in the report concerns the execution of the AND and OR operations in Boolean expressions. It is not defined whether the second term of an expression containing AND will be evaluated if the first term already evaluates to FALSE. Thus in some implementations both terms are always evaluated, in others not. This makes it difficult to write portable programs and also increases the translation effort. The problem is by no means relevant only for expressions containing functions with side effects (which should really be avoided!). Consider:

```
WHILE (I<=MAX) AND (A[I]=0) DO ...
```

In this example, A is supposed to be an array whose upper bound is MAX. At some time, the index I will increase beyond MAX, making I<=MAX false, and then of course A[I] should not be evaluated. It is not at all easy to rewrite the above WHILE without making it look clumsy and rather less readable.

Here is another example which only works correctly if AND "skips":

```
WHILE (P <> NIL) AND (P^.KLASS = VARS) DO ...
```

The compiler itself is full of tricky REPEAT constructs with flags in order to avoid exactly this problem. I believe that the AND and OR operators should always "skip" and that they should evaluate the operands in the order of writing.

type compatibility: there are some very subtle problems with the type-definition mechanism and only one will be shown here. Most Pascal compilers of today compare types by structure. The Report gives no hint as to how one decides that two objects are of the "same" type, but the examples in the User manual always do it by identifier. Type identity by identifier is also preferred by the ISO draft standard. However, if we actually do what most existing compilers claim to do, then the two types A and B shown here are equivalent:

```
TYPE A = RECORD  F: T;  N: ^A END;
     B = RECORD  F: T;  N: ^A END;
```

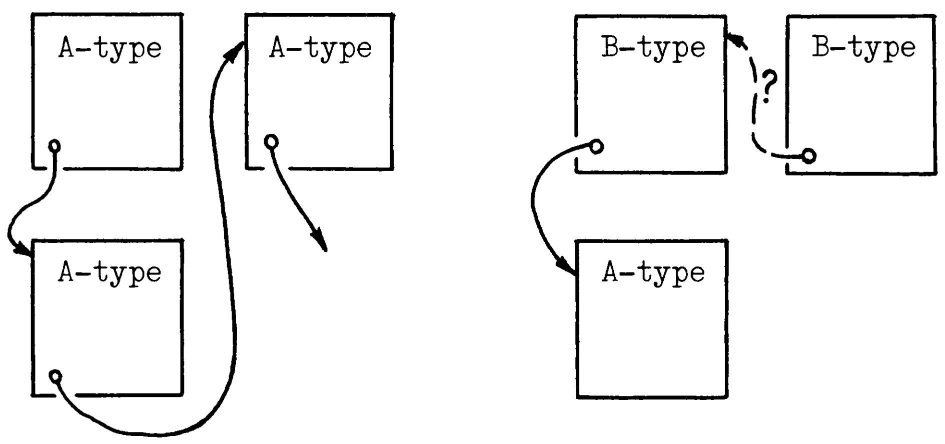


Fig. 21

But as far as probable intentions of the programmer are concerned, I would say that A would be used to make lists and trees whereas B would be used to gain access to the A-structures.

As to the reactions of compilers: some accept the declarations of A and B, some produce an error, some simply blow up!

files: I should mention the problems with sequential file handling as defined by Pascal. Unfortunately, nearly all examples with files are rather complex, so I shall have to skip over this area except for the following remarks:

- there are no procedures for gaining access to files which are specified only at run time. This is understandable, as all operating systems have different ways of accessing files by name, and impose different file name syntax. It is thus impossible to write a portable editor (say) because manipulation of files with respect to opening and closing is totally absent. Let us note that this is not particularly a Pascal problem...

- the standard states that at program startup, the buffer variable of a file will contain the first element of the file (if any) and that the values of the functions EOLN and EOF will be defined. This is the only exception to the general rule that all variables have totally undefined values in the beginning. It essentially prevents the use of Pascal for the writing of interactive programs. Therefore all useful implementations have come up with some sort of remedy for files connected to terminals. It is also a great nuisance to try and implement the "feature" of automatic buffer initialisation if one permits the creation of temporary files, such as those declared inside a procedure.

- the statement

  ```
  READ(F, CH)
  ```

  is, of course, defined to be the equivalent of

  ```
  CH := F^; GET(F)
  ```

  and not of

  ```
  GET(F); CH := F^
  ```

  but how do you explain that to people? The profound reasons for this definition are quite obvious to the implementor, but the simple recommendation should be: do not mix GET and READ in the same program.

- there is no syntax for reading and writing of random access files, because systems use too many different ways to implement such files, or so it is claimed. This is a nonsense argument, since Fortran seems to be able to cover more than 90% of these "different ways".

OK now, this would be bad enough (fig. 22), but there is more.

Fig. 22

7.6. Problems in managing the coding

All the problems shown previously could be programmed around with more or less trouble. Now we come to the very important domain of program maintenance and management of software development.

As before, the problems that I will mention are among those which are most easily communicated, they are not necessarily the worst ones and the set is nowhere near complete.

7.6.1. Compile time expressions

In fig. 23, some uses of the constants N and M are shown.

```
CONST N = 10;  M = 20;
      SIZE = N*M;

VAR A: ARRAY [1..N*(M-1)] OF INTEGER;

VAR LINECOUNT: INTEGER;
    VALUE LINECOUNT (0);     <-- only in 1969 version

TYPE PERSON = RECORD
                NAME: ARRAY [1..2*N] OF CHAR;
                AGE:  0..150;
                SEX:  (MALE, FEMALE)
                END;

VAR EMPTY;

...
EMPTY := PERSON('                    ',0,MALE);
...
```

Fig. 23

Most of these examples are forbidden, because they involve operators, which would have to be applied at compile time. Not only is it impossible to let the value of a constant depend on some previously declared ones, it is also, and more importantly, impossible to use constant value expressions in statements or as initial values. (the VALUE statement in fig. 23 is not standard, but an addition that many implementors have made, or have kept from the original version which permitted it).

Thus parmeterizing of program features is difficult and very unreliable. (A ridiculous example can be found on page 54 of the manual [13]). Most assemblers do a lot better!

7.6.2. Library functions

Generalized functions, as used to provide a general service to their users, are impossible at present. The reason is that Pascal´s strict type checking mechanism does not permit the passing of parameters other than those which match exactly. The most commonly mentioned example of the problem is that of array parameters with differing bounds. In fig. 24, the function LENGTH can be made to work on SHORT or LONG strings, but not on both (it is shown on SHORT).

```
CONST SHLENG = 20;  LOLENG = 100;
TYPE  SHORT = ARRAY [1..SHLENG] OF CHAR;
      LONG  = ARRAY [1..LOLENG] OF CHAR;

VAR   X,Y,Z: SHORT;
      U,V,W: LONG;

FUNCTION LENGTH(S: SHORT): INTEGER;
   VAR I: INTEGER;
   BEGIN
   I:=SHLENG;
   WHILE (I>2) AND (S[I]=´ ´) DO I:=PRED(I);
   LENGTH := I
   END;
```

Fig. 24

Thus we must write two copies of LENGTH, with different names too!

The ISO draft standard solves this problem elegantly [12]. But, although the solution is most appreciated, it works only for arrays. The problem of what to do with records that share common sets of fields remains whole. Thus it is still impossible to write some general list & tree processing procedures....

7.6.3. Separate compilation

So far, we have talked only about problems encountered at the level of the actual source program text. If we now look at what happens during the development or maintenance of a large program or system of programs, then we observe that for a long time people have successfully been using the techniques of modularisation, separate compilation, construction of libraries of common routines etc.

Fig. 25 shows a large program.

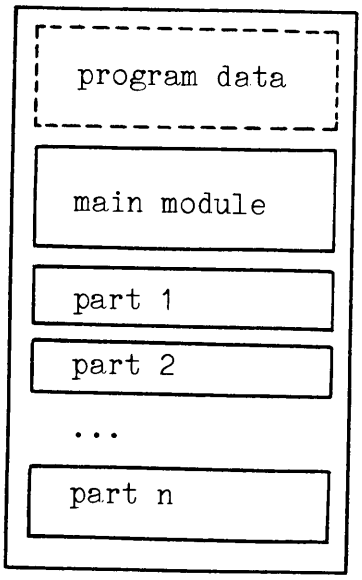


Fig. 25

Clearly here is a case for separate compilation. Note that in this example the parts and the main module must operate on a set of common data, and that it must therefore be possible to access these data from separately compiled parts.

Now here is another situation (fig. 26) in which several programs are shown, each using the services provided by a package.

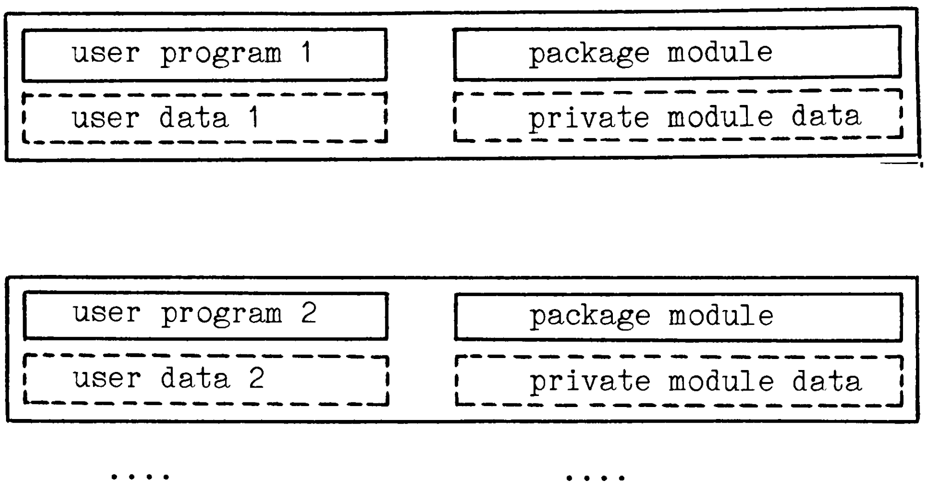


Fig. 26

In this case, the package needs to know nothing about the calling program´s data, but it may want to have its own local data.

of procedures must be able to have access to both global program data and private module data. (Note: the private data must not be local to a procedure but must be remanent, like Algol60 OWN-variables).

Neither possibility exists in Pascal since the notion of separate compilation is absent (even from the ISO draft standard).

Most compilers allow for the separate compilation of procedures, some allow these procedures to have access to the program´s global variables. To my knowledge only the PS compiler allows the programmers to construct modules with both types of data. For a useful language, this is of vital importance.

A third case for separate compilation is that in which several programs run concurrently and are synchronised by signals (fig. 27).

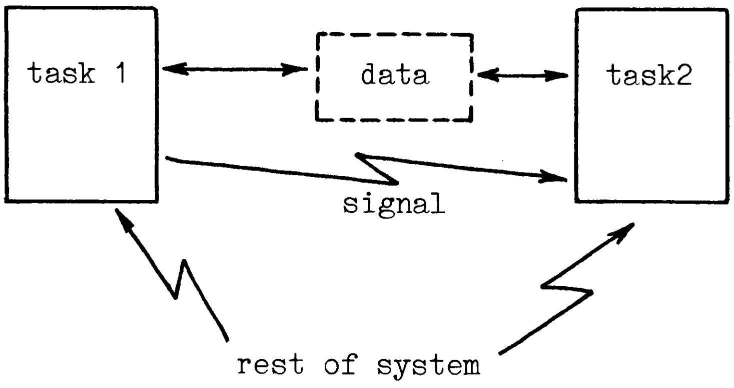


Fig. 27

This is what happens in all "real-time" systems. Pascal, as a purely sequential language, ignores the issue completely. Note that this is not worse than Fortran.

Note that I have not touched the problem of how to check syntactically that the interfaces in separately compiled modules correspond.

8. Useful application areas and conclusion

After having cracked down a lot on Pascal, let me tell you that in spite of all these minor and major shortcomings, I still believe that we have today on the Nord computers no better programming language for general applications.

With some minor changes to sequential file access, Pascal is definitely very useful in the following areas:

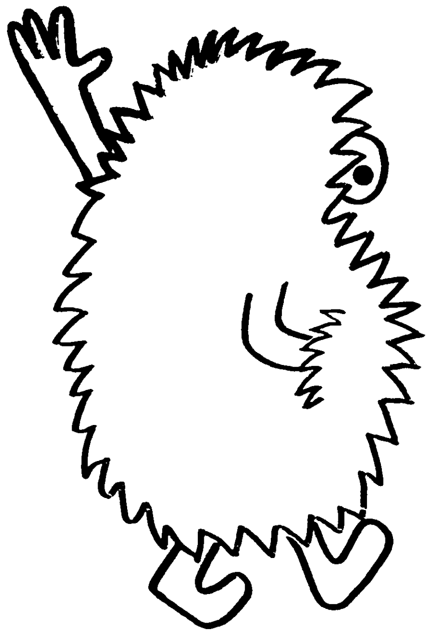
- compiler writing, cross assemblers & compilers...
- text processing
- general, off-line utility programs (editors, etc.)
- treatment of non-numerical data
- processing of trees, lists and other complex data structures
- some mathematical problems
- teaching
- construction of portable programs.

Without extensions, it is not suited for:

- systems programming
- real time and process control
- parallel processing
- construction of large programs and/or packages
- simulation
- numerical analysis
- handling of random access files.

At the PS division, we are continuing to use it, and several of our programs have been distributed to outside users. We have made some changes to be able to write real time applications in our very specific environment, and this has worked well. It is therefore only fair to end this talk with the remark:

> It is grossly unfair to judge an engineering project by standards which have been proved attainable only by the success of the project itself, but in the interests of progress, such criticism must be made.
>
> J. Welsh, W.J. Sneeringer, C.A.R. Hoare.
> 1977 [15]



9. References

[1] The programming language Pascal and its design criteria
N. Wirth, Infotech State of the Art report nr.7, oct. 1969
Infotech information Ltd., 1972

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

[3] The design of a Pascal Compiler
N. Wirth, Software Practice and Experience, 1, 309-333 (1971)

[4] Critical comments on the programming language Pascal
A.N.Habermann, Acta Informatica, 3, 47-57 (1973)

[5] The programming language Pascal, revised report.
N. Wirth, ETHZ report, July 1973

[6] Reply to a paper by A.N.Habermann on the programming language Pascal
O.Lecarme, P.Desjardins, Sigplan Notices, October 1974

[7] An assessment of the programming language Pascal
N. Wirth, Sigplan Notices, June 1975

[8] experience with Pascal on minicomputers
D. Bates, R.Cailliau, Sigplan Notices, Nov. 1977

[9] PS Pascal User´s guide
D.Bates, R.Cailliau, Cern internal note 1976.

[10] Pascal user´s guide,
T. Noodt, Norsk Data A.S. 1979

[11] BSI Draft Pascal standard for Public comments
British standards Institution 79/60528 DC, 1979

[12] A draft proposal for Pascal
A. Addyman, Sigplan notices, April 1980

[13] Pascal, User manual and Report
K.Jensen, N. Wirth, Springer Verlag, 1978

[14] Operator Identification in ADA
K. Ripken et al., Sigplan Notices, April 1980

[15] Ambiguities and insecurities in Pascal
J.Welsh, W.J.Sneeringer, C.A.R.Hoare

[16] PLANC user´s guide
Norsk Data A.S. 1980