# SOME IMPROVEMENTS OF ISO-PASCAL

J.F.H. Winkler
Siemens AG ZTI SOF 213
Otto-Hahn-Ring 6
D-8000 München 83 West Germany

## 1 INTRODUCTION

The programming language Pascal [Wi 71a; JW 75] was a significant contribution to the evolution of programming languages. It is a simple but powerful language in the tradition of Algol 60 [Nau 63]. Compared with Algol 60 it contains more mechanisms for the definition of data structures: enumeration types, records, pointers, and I/O. Algol 60 contains only the array schema which is also included in Pascal. These data structuring facilities are the main reason for the usefulness of Pascal as a language for formulating general algorithms. Some more complicated mechanisms of Algol 60 were left out of Pascal : call by name, switch, dynamic arrays, and the general for-loop.

The proliferation of Pascal was also facilitated by the highly portable Pascal compiler developed at the ETH Zürich. This compiler system consisted of two parts : a compiler from Pascal to P-code (an instruction code for a simple stack computer) written in Pascal and the same compiler written in P-code. To implement Pascal on a new computer it was only necessary to make a P-code interpreter or code generator for the new machine.

Originally Pascal was a language intended for teaching and learning programming. But since it was a very useful language and easy to implement it was and is also used in the commercial field for writing system and application software. For this use the scarce set of mechanisms (which was quite right for teaching and learning) was not sufficient. This led to the situation that different implementors implemented different extensions (e.g. dynamic arrays, random access files, data bases, independent compilation, etc.). Furthermore this compromised portability which was traditionally good for pure Pascal since most compilers were derived from the Zürich compiler. To overcome this situation a group of BCS started work on standardization of Pascal in 1976 [Add 80]. Through numerous drafts and discussions this led to the Draft ISO Standard 7185 [ISO 82].

This draft standard describes essentially the language as it was defined by Wirth [JW 75; Wi 71a]. From the beginning of the standardization effort it was a goal not to revise the language [Add 80] but only to establish a more precise definition of it since the report in [JW 75] was felt inadequate and vague [Add 80; BF 80; Rav 79]. This guideline was followed in the work which led finally to [ISO 82].This document contains one extension and some restrictions of the language but is much more precise and explicit than the report in [JW 75]. The extension consists of a sort of dynamic array called conformant array schema. It can be used as parameters of subprograms.

The decision to standardize Pascal "as it is" retains deficiencies of the language which were seen already by Wirth [Wir 75a]. Furthermore new problems are introduced by the extension and restrictions mentioned above and by the attempt to define some concepts more precisely.

The following paper discusses only features contained in [ISO 82]. We do not look at Pascal with the eyes of someone requiring a language with much more concepts [Pos 79].

Since the future goal of the standardization committee is the standardization of an enriched Pascal the paper may be useful for this undertaking.

The body of the paper contains a number of programs resp. program fragments. They should not be misunderstood in a way that they express a recommended style of programming. Their main purpose is to show the difficulties a processor (processor in the general sense used in [ISO 82]) has while checking a program for conformity with the rules laid down in [ISO 82].

## 2 SCOPE RULE

The scope rule in [ISO 82] is the same as in [JW 75] (see e.g. [Sal 79]). This rule can be characterized by the following three properties :

s1) declaration precedes application

s2) scope = block

s3) inner blocks with redeclarations are <u>totally</u> excluded from the scope of an outer declaration.

At first glance these three properties seem to characterize a sound scope rule. But unfortunately property s2 has consequences which indicate that it should be modified. The problem comes from the fact that the scope of a declared entity extends from the beginning of the block the declaration is immediately contained in until the end of this block, and may thus encompass a piece of program text preceding the declaration itself.

A very popular example for a program which violates this scope rule is depicted in (2-1).

```
(1)    const A = 10;
                 . . .
(2)    procedure P;
(3)        const B = A;                        (2-1)
                  . . .
(4)              A = 15;
                  . . .
(5)    end  { P } ;
```

According to s3 the A in line 3 cannot refer to the outer A declared in 1 because the whole block of P is excluded from the scope of this outer A. This is a consequence of the redeclaration in line 4. Since the applied occurence in line 3 precedes the defining occurence in line 4 the program is illegal (s1).

In order to indicate this violation of the Pascal scope rule all declarations from outer blocks used in the declaration part of an inner block must be marked or imported in the inner block [BF 80a]. This results in a complication of the compiler.

A more complicated example of the scope rule is depicted in (2-2).

```
function F;
      . . .
   function F;
      . . .                                (2-2)
   end { F } ;
      . . .
begin  { body of the outer F }
   F := 10;  { illegal !! }
```

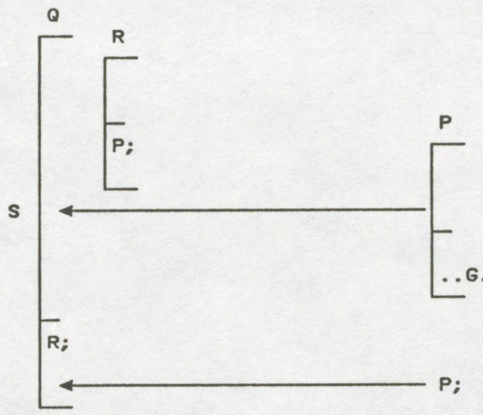
It is impossible to define the value of the outer function because its block is excluded from the scope of its declaration. Apart from the scope problems this may be an indication that the assignment form is not well suited for the definition of the result of a function. The return-clause as used in PL/I [ECM 76], Ada [Ref 83] and Modula-2 [Wir 78, 80] or the result-clause and return-clause of CHILL [ITU 81] are preferable solutions.

A further complication is shown in (2-3) [BF 80].

```
procedure P0;
      . . .
end { P0 };
procedure P1;
      . . .
   procedure P2;
   begin
         . . .                             (2-3)
      P0;
         . . .
   end { P2 };
      . . .
   procedure P0;  { illegal !! }
         . . .
      end { P0 };
         . . .
   end { P1 };
```

The example (2-3) shows that an outer declaration which is used in an inner block must also be imported in all blocks hierarchically between these two blocks.

The general problem with the scope rule of [ISO 82] can best be seen in the following example, where a procedure-declaration P, which contains a global reference G, is inserted at the point S into a procedure Q (2-4). Since Pascal has a monolithic program concept such an insertion of "building blocks" will often be necessary during the development of large programs.



(2-4)

A sound condition that P may be inserted at point S would be :

a) there is no immediate name clash within Q, i.e. no declaration for P is already immediately contained in Q.

b) the global references of P can be associated with corresponding declarations in Q (before the point S) or in the program part before Q.

If we assume that these two conditions hold in (2-4) and that R does not contain a local declaration for P the rule "scope = block" has the consequence to invalidate Q by the insertion of P. This follows from the fact that the applied occurrence of P inside of R precedes the new defining occurence of P inside of Q, after the insertion of P has been carried out. This situation is not easy to understand, and should not be part of a language intended text model" of Ada [Ref 83] is preferable. In this model property s2 is replaced by "the scope begins with the

declaration and ends with the end of the immediately embracing block" and s3 by "parts of inner blocks with a redeclaration are excluded corresponding to property s2)".

The best solution for these scope problems seems to be to forbid the global view [SW 73] and to use explicit import clauses for global entities.

A further complication arises if the declarations of two (or more) entities are mutually dependent as in the case of mutual recursive subprograms or self-referencing data types. The problem of mutually dependent subprograms is resolved in [ISO 82] by the compiler directive FORWARD. The solution of the problem of self-referencing data types is achieved by an exception of rule s1 : "namely that an identifier may have an applied occurrence in the type-identifier of the domain-type of any new-pointer-types contained by the type-definition-part that contains the defining-point of the type-identifier" [ISO 82: 6.2.2.9].

Therefore the following program fragment (2-5) conforms to the requirements of the Draft Standard.

```
(1)    type A = 1 .. 10;
           . . .
(2)    procedure P;
(3)       type B = ↑A;
           . . .                                          (2-5)
(4)         A = record
(5)               K1 : CHAR;
(6)               K2 : B;
(7)             end;
```

In this situation the applied occurrence of A in line 3 must be associated with the declaration starting in line 4.

In order to avoid the above mentioned exception of rule s1 the same mechanism as for subprograms could be used. This would simplify the language and is shown in (2-6).

```
type A = 1 .. 10;
       . . .
procedure P;
   type A = forward;                                  (2-6)
        B = ↑A;
       . . .
        A = record
              K1 : CHAR;
              K2 : B;
            end;
```

(2-6) shows quite instructively that the <u>forward</u>-declaration separates the problem of mutually dependent entities from the scope problem since in (2-6) there is no potential reference to the outer A as in (2-5).

With this solution rule s1 holds without any exception for all conformant programs. The compiler could give a warning or a hint if a type is unnecessarily declared as <u>forward</u>. The <u>forward</u>-declaration is only necessary for a situation as depicted in (2-6); the situation in (2-1) can easily resolved by a simple renaming. But this is not sufficient for (2-5). For the solution (2-6) to work in every case it is necessary that <u>forward</u> is a word-symbol. This is preferable to the solution in [ISO 82] were FORWARD is a so called compiler directive. If <u>forward</u> is a word-symbol the concept of a directive is no longer necessary for the definition of the language since the only required directive in [ISO 82] is FORWARD. Thus the language becomes simpler and easier to learn.

In general rule s1 can only be fulfilled by using some sort of forward construct or by forbidding the declaration of entities refering to each other.

## 3 LOOP VARIABLE

The Draft Standard [ISO 82] states that

l1) the loop variable of a for-loop must be declared in the immediately surrounding block;

l2) no assignment to the loop variable is allowed during the execution of the body of the loop.

l3) neither a loop-body nor the declaration-part, which contains the declaration of a loop variable I immediately, shall contain a statement which may assign a value to I. There are four kinds of statements S which may assign a value to I:

- S is an assignment statement with I as left hand side;
- S contains I as an actual var-parameter;
- S contains I as a parameter to one of the required procedures READ or READLN;
- S is a for-statement and the control-variable of S denotes I.

Rule l3, which is quite restrictive and complicated, is necessary to make the enforcing of rule l2 by the compiler feasible. The language could be made simpler if a loop construct were chosen such that the first occurrence of the loop variable in the for-list acts as the declaration of a constant local to the loop as in Algol 68 [FKL 76; HM 80] and Ada [Ref 83] . The loop would then form a scope by itself. Such a solution is also suggested by the results reported in [PS 79a].

# 4 ARRAY TYPES AND PARAMETERS

## 4.1 THE ACTUAL SITUATION

The introduction of a new sort of formal parameter, the conformant array parameter (CAP), is the main extension of [ISO 82] in comparison to [JW 75]. It was also the main problem which delayed acceptance of a final standard by the ISO [Min 80]. This extension was only included in the Proposed Standard after "two critical pressures were applied to the Committee by N.Wirth and C.A.R. Hoare (independently)" [Sal 80a: 54].

An example for a CAP is :

```
P : array [u .. v : IT] of ET
```

The part after the colon is called a conformant array schema (CAS).

IT is an identifier of an ordinal type. It indicates the index type of actual parameters admissible for the formal parameter P. The identifiers u and v denote the smallest resp. largest value of the actual parameter's index type. These values must always be values of IT.

ET is a type identifier or another CAS and indicates the element type of actual parameters admissible for the formal parameter P.

The CAP removes a significant technical deficiency of Pascal which had already pointed out by Wirth [Wir 75a: 26]. This deficiency was also a subject of critical papers on Pascal [Hab 73 :48; Con 76 : 15]. The reason for this criticism is that it was e.g. impossible to write library procedures for matrix handling in Pascal [BJ 81: 215; DL 75; EH 82]. As a consequence of the extremely static view of data types and objects a Pascal subprogram could only accept arrays of a fixed size as actual parameters.

The description of the new kind of parameter - the conformant array parameter- in [ISO 82] is quite complicated and it is questionable if this is adequate in a language "suitable for teaching programming".

The essential program properties related to the CAP are :

c1) every conformant array schema (CAS) defines a new

type different from all other types in the program.

c2) a CAP cannot be used as an actual parameter in a subprogram call.

c3) a CAP is not assignment compatible to a local array variable with the same bounds and the same element type.

c4) assignment statements involving arrays need no dynamic checks on index ranges since assignment compatible array types have always the same number of elements. This does also hold for CAP.

c5) the length of an activation record for any subprogram activation may be determined at compile time.

Property c2 is the most troublesome of these properties since it hinders the writing of modular programs using CAP. It is e.g. not possible to write a subprogram P which has a CAP and gives this CAP in a call to another subprogram (e.g. a library element) as is shown in (4-1).

```
function F (P:array[u .. v:IT] of ET): FT;
          . . .
   ...    LibFunc(P) ...    {illegal}               (4-1)
          . . .
end  { F };
```

There is only one reason in favor of property c2 given in [ISO 82] : the length of the activation record for any subprogram call can be determined at compile time (property c5). Property c5 does not seem to be very important since the heap storage of a Pascal program can generally not be determined at compile time. There are arbitrary sequences of allocations and deallocations possible. Compared with this highly dynamic structure of the heap it seems not important that the length of the stack frames can be computed in advance. It must also be noted that the number of stack frames cannot be computed at compile time. It seems more natural to look at the whole data storage of a Pascal program as a segmented memory like the heap [Mar 79].

It is also possible to retain the property of fixed length of stack frames if the stack contains only anonymous pointers to the dynamic arrays which are located on the heap [EH 82].

## 4.2 TOWARDS A NEW SOLUTION

A practically useful array feature sould have the following properties :

a1) dynamic array parameters can be used as actual parameters;

a2) there are dynamic array objects;

a3) the declaration of objects assignment compatible to a dynamic array parameter is possible.

Ad a1:
the reason for this property is given in (4-1). This property has the consequence that property c5 does no longer hold.

Ad a2:
the CAS does not allow to write a program, which reads a number N and then declares an array with the index range 1 .. N. This is typical for programs solving problems with varying problem sizes e.g. the solution of systems of linear equations where the number of equations and/or unknowns may vary. Using the CAS concept has the consequence that there must always be a static array which may then be used as an argument for a dynamic array parameter. This is essentially the same situation as in Fortran [FOR 78]. Thus it is not possible to write programs which may adapt themselves to varying problems sizes. Introducing dynamic arrays in this more general sense is not very expensive if a feature like CAS is already in the language since the CAS concept requires descriptors for arrays and the mechanisms for the addressing of such arrays. Thus there are no additional mechanisms necessary to realize this more general array feature. Only property c5 does no longer hold.

Ad a3:
this property is essential for algorithms involving sequences of arrays e.g. iterative solution of systems of linear equations or iterative computation of eigenvalues [SS 76]. After one iteration usually an assignment statement of the form "Old_Array := New_Array;" will be executed. The size of these arrays could be given by a dynamic array parameter. If the CAS concept is used such an assignment is only possible if Old_Array and New_Array are both declared in the same CAS. This means that an internal auxiliary variable must be supplied from outside as an actual parameter. This would lead to an unacceptable programming style. If an internal auxiliary variable is used the above assignment can only be realized by a loop statement. This would lead to inefficient code on most machine architectures.

The main problem now is how the properties a3 and c4 can both be fulfilled. If we use a similar scheme for dynamic array parameters as in [ISO 82] the type of these parameters will be an anonymous type. There will be no type identifier to indicate the type of the local object which should be assignment compatible (in the sense of c4) to the dynamic array parameter. The following proposal uses the <u>like</u> clause to indicate that the properties of a new object are to be the same as those of an already declared <u>object</u>.

The proposal consists of three parts : the introduction of dynamic array types, dynamic array parameters, and the <u>like</u> clause[1].

## 4.3 DYNAMIC ARRAY TYPES

```
array-type = "array" "["index-range (","  index-range)"]"
             "of" component-type.
index-range = ordinal-type-identifier |
              ordinal-expression ".." ordinal-expression.
ordinal-expression = expression.
```

The new feature in this proposal is the production

index-range = ordinal-expression ".." ordinal-expression. (4-2)

which allows arbitrary ordinal-expressions for the bounds of index-ranges. Both expressions in one index-range must be of the same type. The expressions of an index-range are evaluated in an arbitrary order. Their values are the lower resp. the upper bound of the index-range. If the uppper bound is a predecessor of the lower bound the index-range is empty. The index-ranges of an array-type are elaborated in an arbitrary order.

If at least one of the two ordinal-expressions in (4-2) cannot be determined at compile time the corresponding index-range constitutes a dynamic index-range. An array-type which contains at least one dynamic index-range is a dynamic array-type.

The rules for assignment compatibility are the same as in [ISO 82]. As a consequence of this property c4 does also hold in our proposal.

The production (4-2) is only necessary because we do not assume the existence of dynamic ordinal-types. If there were dynamic ordinal-types in the language the production

[1] A <u>like</u> clause is contained in the earlier version of PL/1 [IBM 72: 345]. But this clause was not incorporated into the PL/I standard [ECM 76].
Independent of our paper here, a <u>like</u> clause was just recently proposed in [Wha 83].

```
index-range = ordinal-type-identifier
```

alone would be sufficient.

There are a lot of proposals for dynamic arrays in the literature [EH 82; Kit 77; Mac 75; Mof 81; Pok 76; Ste 76; Ten 83; Wir 76a]. The solution in this proposal is similar to the scheme already used in Algol 60 [Nau 63] and was also proposed by Wirth [Wir 76a].

## 4.4 DYNAMIC ARRAY PARAMETERS

```
formal-parameter-section >
        dynamic-array-parameter-specification.

dynamic-array-parameter-specification =
        ["var"] identifier-list ":" "dyn" ["packed"]
        static-array-type-identifier.
static-array-type-identifier = type-identifier.
```

A formal dynamic-array-parameter-specification has essentially the same structure as other data parameter specifications [Ten 80: 11] :

```
["var"] id-list ":" type-indication .
```

This means that the language becomes simpler and easier to learn by this proposal. Instead of

<u>var</u> PAR : <u>array</u> [L .. U] <u>of</u> ET

we simply write

<u>var</u> PAR : <u>dyn</u> AT

AT would be a one dimensional array type with element type ET. In general AT can denote any (static) array-type which has been declared previously and is visible at this point.

In a dynamic-array-parameter-specification a new (anonymous) array-type is introduced whose properties are derived from an already declared (static) array-type.

Conformability is defined as in [ISO 82: 6.6.3.8].

The definition of parameter compatibility is much more simpler than in [ISO 82] :

pc) an array A may be an argument for a formal dynamic array parameter D iff the type of A conforms to the type of D.

Intuitively this means that an actual array is "not greater" than the formal, and the element types of both are the same.

The rule above allows also the use of a dynamic array parameter as an argument for another dynamic array parameter (even for itself in the case of a recursive subprogram).

In contrast to [EH 82; Pok 76] we have introduced a new sort of parameter which is simply indicated by "dyn". This tells the reader of the program directly what the properties of the corresponding parameter are and seems to be much more simpler than the concept of parametric types [EH 82].

Inside a subprogram containing a dynamic array parameter in its heading it is necessary to know the index bounds of the corresponding actual-parameter. In this proposal two families of required functions are introduced for this : LBound and HBound. They are defined for any array type and array object. Let t denote an array type or an array object. LBound(t) (HBound(t)) yields the lower (upper) bound of the first index-range of t. Let n be a static integer expression yielding a number greater than zero. LBound(t,n) (HBound(t,n)) yields the lower (upper) bound of the n-th index-range of t. It is an error if the value of n exceeds the number of index-ranges of t.

An example for this proposal is given in figure 4-1.

```
   type AT1 = array [1..10] of INTEGER;
        AT2 = array [1..100] of AT1;

   var  AV1 : array [1..100] of AT1;
        AV2 : array [50..60] of array [1..5] of INTEGER;

   procedure P (var  A : dyn AT2);
      var I : LBound(AT2,1) .. HBound(AT2,1);
                   {since AT2 must be a static array type
                    the type of I is also static }
          J : LBound(AT2,2) .. HBound(AT2,2);
   begin
      for I := LBound(A,1) to HBound(A,1)
                   { the iteration clause comprises exactly the
                    index values of the actual parameter }
      do begin
         for J := LBound (A,2) to HBound(A,2)
         do A[I,J] := A[I,J]  + 7;
         end;
   end;
     . . .
   P(AV1);
   P(AV2);
```

Figure 4-1. The use of dynamic array parameters

This approach to the problem of adjustable array parameters also avoids the main problems pointed out in [Min 80: 79..83]. The main reason for this are the two newly introduced (families of) required functions "LBound" and "HBound" which are used instead of the "bound-identifiers" in [ISO 82]. These functions lead to a better semantic coherence of the program, in that a semantic dependency between several items (the object A and its index bounds) is used to compute the dependent value from the independent one.

The example (4-1) contains one remaining insecurity: LBound(A,1)..HBound(A,1) may be a proper subset of LBound(AT2,1)..HBound(AT2,1). This insecurity can be removed by using the proposal for the implicit declaration of the loop variable (conf. sect. 3). This leads to the program fragment (4-2) where the set of admissible values for I and J are (automatically) constrained as far as possible.

```
     type AT1  .....
          {as in figure 4-1}
     procedure P (var A: dyn AT2);
          { no declaration for I and J }
     begin                                                  (4-3)
        for I := LBound(A,1) to HBound(A,1)
        do begin
           for J := LBound(A,2) to HBound(A,2)
           do A[I,J] := A[I,J] + 7;
           end;
     end  { P } ;
```

The restriction of the type of a dyn-parameter to a static-array-type is only necessary because we do not assume the existence of dynamic ordinal-types. If there were dynamic ordinal-types in the language the procedure P in figure 4-1 could simply be written as in (4-4). In (4-4) the type AT2 could then also be a dynamic array-type.

```
     procedure P (var A : dyn AT2);
        var I : LBound(A,1) .. HBound(A,1);
           { now I has the proper value range }
            J : LBound(A,2) .. HBound(A,2);                 (4-4)
     begin
           { as in figure 4.1 }
```

## 4.5 THE LIKE-CLAUSE

The declaration of local array variables which are assignment compatible with a formal dynamic array parameter is not possible using the elements defined so far. If the dynamic parameter is defined by "var P: dyn AT;" then "dyn AT" defines a new type different to all other types in the program. A variable V declared locally by the declaration "var V: AT;" is therefore not assignment compatible with P.

Assignment compatibility can be achieved by deriving the properties of the local variable V directly from the parameter itself by the like-clause :

```
                    var V: like P;
```

This means that V has the same type as the actual parameter corresponding to the formal parameter P. As a consequence the assignment

```
                    V := P;
```

is perfectly legal and does not need dynamic checks on the index bounds since V and P have by definition the same number of components.

An example for the use of the like-clause is given in figure 4-5.

```
     type VECTOR = array [INTEGER] of REAL;
               . . .
     procedure P(var OldVector : dyn VECTOR);
        var NewVector : like OldVector;
     begin
        repeat                                              (4-5)
           OldVector := NewVector;
           {computation of
            NewVector := F(OldVector)  }
        until  Cond(NewVector, OldVector);
     end  { P };
```

It is not necessary to restrict the use of the like-clause to dynamic array parameters. Any variable already declared could be used in the like-clause. This could avoid the introduction of type names, when it is not really necessary [Ker 81: 6], and may thus reduce the conceptual load for the writer and the reader of the program [Bak 80]. It does furthermore enhance the semantic coherence of the program because the relationship between two (or more) variables is expressed more directly than by using a type. The like-clause is not intended to replace the type definition facility. It is an alternative which may be better suited for certain situations. In which cases the use of the like-clause is to be preferred to the use of a type can only be decided by practical experience. One situation in favour of the like-clause has been discussed above.

## 5 INPUT-OUTPUT

In the input-output area the problem is the input for interactive programs [FG 82: 88; Kay 80]. Before we describe the problems we will briefly mention the main charcteristics of input-output in Pascal.

Input-output is realized via variables of type "file of ElementType" where ElementType can be any type which does not contain (directly or indirectly) a file type. A file (value) is a linear sequence of values of the ElementType. With each file variable f a so called buffer variable f↑ of ElementType is associated. The buffer variable contains one element of the associated file variable. Only this component can be accessed and manipulated directly. Other elements can only be accessed by scanning the file variable sequentially. In this framework input-output consits of the manipulation of file variables and/or buffer variables. The following predefined procedures and functions may be used for this : RESET, REWRITE, GET, PUT, READ, WRITE, READLN, WRITELN, EOF, EOLN, and PAGE.

For human readable I/O Pascal has a predefined file type called TEXT. A value of type TEXT is a sequence of lines where each line is a sequence of characters containing exactly one end of line marker as the last component. Together with the file type TEXT two procedures READ and READLN for formatted input are defined. Furthermore a required TEXT variable INPUT is defined which may occur in the parameterlist of the program and is intended to be associated with the standard input device. In case of an interactive program the standard input device will usually be the input part of a CRT-terminal.

Most of the above mentioned subprograms for doing I/O embody a prefetch discipline which causes one more component to be transferred as is actually needed. "RESET(FileVar)" puts the FileVar in input mode and places the first element (if it exists) into the associated buffer variable FileVar↑.

```
                    READ(FileVar,ElemVar);
```

is defined as

```
          begin ElemVar:=FileVar↑; GET(FileVar); end .
```

The actual element in the file buffer is assigned to the second parameter of READ and the next element of FileVar is assigned to the buffer variable FileVar↑ by executing the procedure GET.

READLN also embodies this prefetch discipline. It places "the current file position just past the end of the current line in the textfile" [ISO 82: 62]. The naive user would rather expect that READLN (= read line ??) reads just a line from INPUT. Pugh and Simpson also report difficulties of novices caused by READLN [PS 79a].

For the required textfiles INPUT and OUTPUT [JW 75] defined that an implicit RESET(INPUT) resp. REWRITE(OUTPUT) is automatically performed by the system. Neither the user manual nor the report define exactly when this RESET(INPUT) is to be executed. Therefore lazy input [HS 78b] has always been allowed in Pascal. Unfortunately the report left many questions open. The real definition was the compiler distributed from Zurich. This compiler implemented the rule about the implicit RESET(INPUT) in such a way that it occurred before the execution of the first statement of the program.

With this implementation the realization of sound interactive programs is impossible because the program requires some input from the terminal (as a consequence of the prefetch discipline in RESET) before it can give any prompting message to the user [HS 78b: 93]. A sound interactive program works the other way round: it gives first some message to the user, e.g. that it is running at all and is expecting some input, and then reads this input from the user.

To avoid this unsatisfactory situation [ISO 61a: 67] suggests lazy I/O for the program parameters INPUT and OUTPUT. The post-assertions of RESET resp. REWRITE shall hold "prior to the first access to the textfile or its associated buffer-variable".

The main drawback of this deferred input is the complication of the translator and the object program [Per 81: 87]. Each reference to INPUT↑ must check if RESET(INPUT) has already been executed. The program fragment (5-1) shows an example for this.

```
(1)  program PROG(INPUT);
(2)     { Declarations }
(3)  begin
(4)     if Condition                                    (5-1)
(5)     then CharVar := INPUT↑;
(6)     { arbitrary statements
(6)       not referencing INPUT or INPUT↑ }
(7)     CharVar2 := INPUT↑;
```

When the statement in line 7 is executed it must be known wether RESET(INPUT) has already been executed or not. This means that generally each reference to INPUT↑ must check the status of INPUT.

The situation is even worse if the file buffer (INPUT↑) is used as an actual var-parameter. Even the paper [HS 78b], which deals with the problems of lazy I/O, does not offer a solution for this problem. The complete implementation of lazy I/O seems to involve a certain runtime overhead since, a var-parameter must carry with it the information wether it is a file buffer or not, and each reference to this parameter must interpret this information.

Using a discipline without prefetch RESET(INPUT) could be executed without any harm before the first executable statement of the program body and no checks at the referencing points were necessary.

Summarizing this discussion gives the result that the real culprit is the prefetch discipline and not the implicit RESET(INPUT).

## 6 SPECIFICATION OF FORMAL SUBPROGRAMS

In [JW 75] the exact nature of formal subprograms need and can not be indicated in the corresponding parameter specification.

As a consequence the actual parameters belonging to calls of formal subprograms have to be checked dynamically. This can be seen in the program fragment (6-1).

```
    program PROG61(INPUT,OUTPUT);
       procedure P1(Par1: BOOLEAN; procedure Par2);
       begin
          if Par1
          then Par2(10)
          else Par2(10,10);
       end  { P1 };

       procedure P2(Par1: INTEGER);                      (6-1)
       begin
          WRITELN("P2; Par1 = ", Par1);
       end  { P2 };

       procedure P3(Par1, Par2 : INTEGER);
       begin
          WRITELN("P3; Par1 = ",Par1,"Par2 = ",Par2);
       end  { P3 };

    begin  { of program }
       P1(TRUE, P2);
       P1(FALSE,P3);
    end.
```

It is interesting to note that in Algol 60, which has similar facilities for the specification of parameters and does also not allow to describe the types of parameters of formal subprograms, a program analogous to (6-1) would be illegal. The reason for this is that the semantics of the procedure call is there defined by the replacement of the call statement by the (modified) body of the called procedure (copy rule). The modification of the body consists essentially of the replacement of the formal parameters by the actual ones. This modification must "lead to a correct ALGOL statement" [Nau 63: 4.7.5]. This additional condition is not fulfilled in (6-1).

In [ISO 82] a parameter which is a subprogram must be fully specified with all its own parameters and in the case of a function additionally with its result type.

As a consequence programs as (6-1) are no longer possible and no dynamic type checking in the calls of formal subprograms is necessary. This is a real deviation from [JW 75]. Unfortunately this has been done in a somewhat unsatisfactory manner in that the specification of the formal subprogram contains parameter identifiers which are superfluous and disturbing. This is exemplified in (6-2) and (6-3).

```
  program PROG62(INPUT,OUTPUT);
     procedure P1(Par11 : BOOLEAN;
                  procedure Par12(Par : BOOLEAN));
     begin
        Par12(Par11);
     end  { P1 };
     procedure P2(Par21 : BOOLEAN);
     begin
        WRITELN("P2; Par21 = ",Par21);                    (6-2
     end  { P2 };
     procedure P3(Par31, Par32 : BOOLEAN);
     begin
        WRITELN("P3; Par31 = ",Par31," ; Par32 = ",Par32);
     end  { P3 };
  begin   { of program }
     P1(TRUE, P2);
     P1(FALSE,P3);   { this is now illegal }
  end.
```

Inside the procedure P1 the identifier "Par" cannot be used for any purpose and the programmer may be disturbed by the fact that in the heading of P2 the corresponding parameter is named by "Par21". In Pascal the form depicted in (6-3) would be sufficient.

```
  procedure P1 (Par11 : BOOLEAN;                          (6-3)
                procedure Par12 (BOOLEAN) );
```

If the keyword notation for actual parameters [Fra 77; Har 76; Par 78] were allowed in Pascal, as it is e.g. in Ada [Ref 83], the identifier "Par" of this second order formal parameter could be useful. This is shown in (6-4).

```
  procedure P1 (Par11 : BOOLEAN;
                procedure Par12(Par : BOOLEAN) );
  begin                                                   (6-4)
     Par12 (Par => Par11);
  end  { P1 };
```

But for third and higher order formal parameters there is even with key word notation no need for identifiers (6-5).

```
  procedure P1 (Par11 : BOOLEAN;
                procedure Par12(
                   procedure Par21(Par31 : BOOLEAN)));
  begin                                                   (6-5)
     Par12 (Par21 => P2 {as defined in (6-1)});
  end  { P1 };
```

There are only a few new syntax rules necessary to specify second and higher order formal parameters without identifiers [Win 76b: 11; DL 74: 25].

## 7 ORDER OF DECLARATIONS

The rule for the ordering of declarations in [ISO 82] is essentially the same as in [JW 75] and [Wir 71]. This rule states that the declarations of the different kinds of entities must be arranged in each declaration-part in the following order : labels, constants, types, variables, subprograms. There seems to be no sound technical reason for this rule. This rule is especially not necessary for the existence of a one pass compiler. A one pass compiler only requires that any declared entity is declared before its first use ("linear text model", property s2 in section 2).

On the other hand the declaration rule of [ISO 82] has negative effects on the software engineering properties of Pascal [HSW 77]. It hinders the programmer to put together what belongs together.

Despite the fact that Pascal does not incorporate the concept of program module one could "simulate" it within the monolithic Pascal program if there were not this rule for the declaration order. An example for this is given in (7-1).

```
     program P (INPUT,OUTPUT);
        { package PACK1 }
           const  .....
           type   .....
           var    .....
           { subprograms }
        { end  PACK1 }                                 (7-1)

        { package  PACK2 }
             . . .
        { end  PACK2 }
             .
             .
             .
     begin
         { main program }
     end.
```

By removing this artificial restriction on the order of declarations a program structure similar to Ada or Modula-2 [Wir 80] could be achieved. The essential difference to Ada would be that a Pascal program is one compilation unit whereas in Ada each package can be a compilation unit.

A programming style like that depicted in (7-1) was already possible in Algol 60. In this respect Pascal seems to be overly restrictive in its current form.

## 8 STRUCTURED STATEMENTS

Pascal uses the "open" form for most of its structured statements, e.g. the if-statement :

```
     if-statement = if condition
                    then statement                   (8-1)
                  [ else statement ]
```

This forces the user to use a compound statement if the then-part or the else-part of the if-statement consists of more than one statement :

```
     if A < 10
     then begin
             C := A + 5;                               (8-2)
             D := 0;
          end ;
```

The "closed" form of the structured statements, which is incorporated in most languages developed during the seventies, has several advantages. A first advantage is that it leads to a less clumsy formulation. The statement (8-2) would be written as :

```
     if A < 10
     then C := A + 5;                                (8-3)
          D := 0;
     end if    { or fi  }
```

There has been and is a controversary discussion about the form of closing key words for the closed form of structured statements [Bro 76; Ham 80; Hun 81; Knu 74a: 266; Kov 78; Mar 81]. From a purely technical point of view any unequivocal symbol would do it. In comparison to the usual usage of brackets in mathematical formulae the use of symmetric key words ( e.g. fi, od etc.) seems to be quite natural. To a certain degree this can be a matter of taste and other solutions ( end if, end loop, etc. ) may also work well. An additional requirement which belongs to the area of human engineering would be that all word-symbols sould be easily pronounceable. In the following example we use symmetric and pronouncable key words.

A second advantage is the fact that the endings of several structured statements, which are nested, can be clearly discriminated, if different structured statements have different closing symbols. (8-4) shows a nesting using Pascal as it is and (8-5) shows the same nesting using symmetric closing symbols.

```
    else
        begin
           if I > DIGMAX then begin
              ERROR(203);
              VAL.IVAL := 0
              end
           else
             with VAL do begin
                 IVAL := 0;
                 for K := 1 to I do begin                 (8-4)
                    J := ORD(DIGIT[K]) - ORD('0');
                    if IVAL <= (MAXINT - J) DIV 10 then
                       IVAL := IVAL * 10 + J
                    else begin
```

```
                     ERROR(203);
                     IVAL := 0
                     end
                   end;
               SY := INTCONST
               end
           end



       else
         if I > DIGMAX
         then ERROR(203);
              VAL.IVAL := 0;
         else
           fix VAL
           in IVAL := 0;
             for K := 1 to I                              (8-5)
             do J := ORD(DIGIT[K]) - ORD('0');
               if IVAL <= (MAXINT - J) DIV 10
               then IVAL := IVAL * 10 + J;
               else ERROR(203);
                    IVAL := 0;
               fi
             rof
             SY := INTCONST;
           xif
         fi
```

Note the use of the word-symbol <u>fix</u> instead of <u>with</u> in the example (8-5) in order to get a pronounceable inverse (<u>xif</u>). <u>fix</u> .. <u>in</u> seems to reflect the semantics of this language construct quite well.

## 9 COMPATIBILITY WITH THE STANDARD

In this section we discuss to what degree the new features proposed in this paper are compatible with the Draft Standard. The crucial point is how many of the already existing conforming programs had to be modified.

### <u>SCOPE RULE</u>

The scope rule of [ISO 82] is more restricted than the linear text model. The examples in (2-1), (2-3) and (2-4) are not conformant with [ISO 82] but are conformant with the linear text model. Thus <u>no</u> modifications of conformant programs were necessary.

The introduction of the <u>forward</u>-declaration for mutually dependent data types would make the modification of programs containing such entities necessary. This can be seen in the examples (2-5) and (2-6). Also programs which use "forward" as an identifier would be affected. But this should not happen very frequently because "forward" was already a compiler directive in [JW 75].

### <u>LOOP VARIABLE</u>

Only those programs had to be modified in which a for loop is terminated by a goto statement and the actual value of the loop variable is used after this termination. The reason for this is that in [ISO 82] a loop variable can only get a defined value if a loop with this loop variable is executed.

### <u>ARRAY TYPES AND PARAMETERS</u>

No program which conforms to [JW 75] or which conforms to [ISO 82] and does not use the conformant-array-schema would be affected by the new concepts proposed in section 4. If a program contains one of the identifiers HBound and LBound a renaming could be necessary in order to use the proposed required functions HBound and LBound. The lexical unit "like" should not be used as an identifier since it would be a new word-symbol.

### <u>INPUT/OUTPUT</u>

If RESET were defined similar to OPEN, which is contained in most command languages of operating systems, and if READ read just the next component into its actual parameter, then an interactive program, which uses only RESET and READ should not be affected.
The postcondition of READ should imply that the component read is the last component of the left part of the file. This condition holds already in [ISO 82]. The definition of EOLN could remain the same as in [ISO 82]. Note that READLN should not be used in its current form for interactive input.

### <u>SPECIFICATION OF FORMAL SUBPROGRAMS</u>

Only programs which use formal, parameterized subprograms would be affected by this proposal. Note that the formal subprogram is a concept used quite rarely.

### <u>STRUCTURED STATEMENTS</u>

Almost every program would be affected, but an automatic conversion tool could be realized easily.

### <u>ORDER OF DECLARATIONS</u>

<u>No</u> program would be affected since the property "declaration precedes application" holds already in [ISO 82].

## 10 CONCLUSION

The outcome of the standardization effort for Pascal is a very enhanced form of the original language [JW 75]. But there are still some trouble spots in the language, which should be resolved in a future revision of the standard.

The main problems are :

- the scope rules as defined in [ISO 81] are difficult to understand, to learn, and to use. Scope rules oriented towards the "linear text model" are better suited.
- the solution for adaptable arrays as parameters is a little bit clumsy and quite difficult to understand and learn. In this paper a simpler solution with a better functionality is proposed.
- the concept of I/O as defined in the original Pascal is not adequate for expressing interactive programs. It is very regrettable that this flaw of Pascal was not removed during the standardization process. It may be a little disadvantageous if a novice is introduced to I/O by means of this I/O-discipline.
- the specification of formal subprograms has been enhanced but it could have easily done better by using some additional syntax rules.

The main problem with this standardization effort still is the fact that the thing which is being standardized nearly doesn't exist because most of the implementations intended for serious programming (e.g. [Sie 82, 82a; Tex 79; UCS 80]) implement a superset of Pascal. Some often added features are :

- a mechanism for independent compilation
- dynamic arrays
- string handling
- random access files

It cannot be expected that such useful features will not be used in the future. Since they are not in the language portability is still compromised.

## ACKNOWLEDGMENTS

The author thanks M.Sommer and C.Stoffel for a critical review of an earlier draft and valuable suggestions.

## 11 REFERENCES

Add 80 Addyman, Anthony M.
Pascal
= [HM 80: 81..91].

AH 80 Austermühl, Burkhard; Henhapl, Wolfgang
A critical review of PASCAL based on a formal storage model
= [Hof 80: 57..69].

Bak 80 Baker, Henry G. Jr.
A Source of redundant Identifiers in PASCAL Programs
SIGPLAN Notices 15,2(1980)14..16.

BF 80 Baker, T.P.; Fleck, A.C.
Does Scope = Block in Pascal ?
Pascal News #17 (March,1980)60..61.

BJ 81 Boute, R.T.; Jackson, M.I.
A Joint Evaluation of the Programming Languages Ada and CHILL
= [IEE 81: 214..220].

Bro 76 Brown, Robert E.
Toward a Better Language for Structured Programming
SIGPLAN Notices 11,7(1976)41..54.

BWW 76 Wettstein, H.; Becker-Weimann, K.; Winkler, J.F.H.; Wosnitza, H.
Ein modernes, modulares Betriebssystem für Prozeßrechner und seine Generierung
PDV-E71, Gesellschaft für Kernforschung, Karlsruhe Juni 1976.

Cai 82 Cailliau, R.
How to Avoid Getting Schlonked by Pascal
SIGPLAN Notices 17,12(1982)31..40.

Con 76 Conradi, Reidar
Further critical comments on Pascal, particulary as a systems programming language
SIGPLAN Notices 11,11(1976)8..25.

DL 74 Lecarme, Olivier; Desjardins, Pierre
Reply to a paper by A.N. Habermann on the programming language Pascal
SIGPLAN Notices 9,10(1974)21..27.

DL 75 Desjardins, P.; Lecarme, O.
More Comments on the Programming Language Pascal
Acta Informatica 4(1975)231..243.

ECM 76 ECMA - European Computer Manufacturers Association
Standard ECMA-50. Programming Language PL/I

Geneva, December 1976.

EH 82 Hennessy, John; Elmquist, Hilding
The Design and Implementation of Parametric Types in Pascal
Software - Practice & Experience 12,2(1982)169..184.

FG 82 Feuer, Alan R.; Gehani, Narain, H.
A Comparison of the Programming Languages C and PASCAL
Computing Surveys 14,1(1982)73..92.

FKL 76 van Wijngaarden, A.; Mailloux, B.J.; Peck, J.E.L.; Koster, C.H.A.; Sintzoff, M.; Lindsay, C.H.; Meertens, L.G.L.T.; Fisker, R.G. (eds.)
Revised Report on the Algorithmic Language Algol 68
Springer, Berlin usw. 1976.

FL 82 Leblanc, Richard J.; Fischer, Charles N.
A Case Study of Run-Time Errors in Pascal Programs
Software - Practice Experience 12,9(1982)825..834.

FOR 78 ANSI - American National Standards Institute
ANSI X3.9 - 1978. Programming Language FORTRAN

Fra 77 Francez, Nissim
Another Advantage of Keyword Notation for Parameter Communication with Subprograms
CACM 20,8(1977)604..605.

Hab 73 Habermann, A.N.
Critical Comments on the Programming Language Pascal
Acta Informatica 3(1973)47..57.

Ham 80 Hamlet, Richard
A Further Note on Symmetric Keyword Pairs
SIGPLAN Notices 15,5(1980)7.

Har 76 Hardgrave, W.T.
Positional versus Keyword Parameter Communication in Programming Languages
SIGPLAN Notices 11,5(1976)52..58.

HM 80 Hill, I.D.; Meek, B.L.
Programming Language Standardisation
Ellis Horwood Ltd. + Halstead Press, Chichester, New York etc. 1980.

Hof 80 Hoffmann, H.-J. (Hrsg)
Programmiersprachen und Programmentwicklung
Springer, Berlin usw. 1980.

HS 78b Hisgen, A.; Saxe, J.B.
Lazy evaluation of the file buffer for interactive I/O
Pascal News #13 (1978)93..94.

Hun 81 Hunt, J.G.
Bracketing Programme Constructs
SIGPLAN Notices 16,4(1981)64..67.

IBM 72 International Business Machines Corp.
IBM System/360 Operating System
PL/1 (F) Language Reference Manual
Order No. GC28-8201-4, 5.ed. Dec. 1972.

IEE 81 IEE - Institute of Electrical Engineers
Fourth International Conference on Software Engineering for Telecommunication Switching Systems.
IEE, 1981.

ISO 82 ISO - International Organization for Standardization
Programming Languages - PASCAL, ISO/DIS 7185, 1982-08-12.

ITU 81 ITU - International Telecommunication Union
CCITT High Level Language (CHILL) - Recommendation Z.200
Geneva, 1981.

JW 75 Jensen, Kathleen; Wirth, Niklaus
PASCAL. User Manual and Report
Springer, Berlin usw. 1975.

Kay 80 Kaye, Douglas R.
Interactive Pascal Input
SIGPLAN Notices 15,1(1980)66..68.

Ker 81 Kernighan, Brian W.
Why Pascal is Not My Favorite Programming Language
CS-TR No.100, Bell Laboratories, Murray Hill
July 18, 1981.

Kit 77 Kittlitz, Edward N.
Another Proposal for Variable Size Arrays in PASCAL
SIGPLAN Notices 12,1(1977)82..86.

Knu 74a Knuth, Donald E.
Structured Programming with goto Statements
Computing Surveys 6,4(1974)261..301.

Kov 78 Kovats, T.A.
Program Readability, Closing Keywords and Prefix-Style Intermediate Keywords
SIGPLAN Notices 13,11(1978)30..42.

Mac 75 MacLennan, B.J.
A Note on Dynamic Arrays in PASCAL
SIGPLAN Notices 10,9(1975)39..40.

Mar 79 Marlin, C.D.
A Heap-based Implementation of the Programming Language Pascal
Software - Practice & Experience 9(1979)101..119.

Mar 81 Marca, David
Some Pascal Style Guidelines

SIGPLAN Notices 16,4(1981)70..80.

Min 80 Miner, Jim
Pascal Standard : Progress Report
Pascal News #19 (Sept 1980)74..84.

Mof 81 Moffat, David V.
Conformant Arrays and Strong Typing
ACM Annual Conference 1981, 161..163.

Nau 63 Naur, Peter (ed.)
Revised Report on the Algorithmic Language Algol 60
CACM 6,1(1963)1..17.

Par 78 Parkin, Rodney
On the Use of Keywords for Passing Procedure Parameters
SIGPLAN Notices 13,7(1978)41..42.

Per 81 Perkins, Hal
Lazy I/O is not the answer
SIGPLAN Notices 16,4(1981)81..88.

Pok 76 Pokrovsky, Sergei
Formal Types and Their Application to Dynamic Arrays in Pascal
SIGPLAN Notices 11,10(1976)36..42.

Pos 79 Posa, John G.
Pascal people unhappy over standard
Electronics (Feb 15, 1979)96.

PS 79a Pugh, J.; Simpson, D.
Pascal errors - empirical evidence
Computer Bulletin 2,19(1979)26..28.

Rav 79 Ravenel, Bruce W.
Toward a Pascal Standard
Computer 12,4(1979)68..82.

Ref 83 Reference Manual for the Ada Programming Language.
ANSI/MIL-STD 1815A.
United States Department of Defense, January 1983.

Sal 79 Sale, Arthur
Scope and Pascal
SIGPLAN Notices 14,9(1979)61..63.

Sal 80a Sale, A.H.J.
Conformant Arrays in Pascal
Pascal News #17 (March 1980)54..56.

Sie 82 Siemens AG
Programmiersystem PASCAL. Benutzerhandbuch Teil 1, Sprachbeschreibung.
Best. Nr. U685-J-Z55-1, January 1982.

Sie 82a Siemens AG
Programmiersystem PASCAL. Benutzerhandbuch Teil 2, Bedienungsanleitung
Best. Nr. U964-J-Z55-1, July 1982.

SS 76 Schmeißer, Gerhard; Schirmeier, Horst
Praktische Mathematik
Walter de Gruyter, Berlin usw. 1976.

Ste 76 Steensgaard-Madsen, J.
More on dynamic arrays in PASCAL
SIGPLAN Notices 11,5(1976)63..64.

SW 73 Wulf, W.; Shaw, Mary
Global Variable Considered Harmful
SIGPLAN Notices 8,2(1973)28..34.

Ten 83 Tennent, R.D.
An Alternative to Conformant-Array Parameters in Pascal
SIGPLAN Notices 18,10(1983)38..43.

Tex 79 Texas Instruments Inc.
Model 990 Computer. TI Pascal User's Manual
Part. No. 946290-9701 *A, 1 July 1979.

UCS 80 UCSD Pascal - Version II.0. USER'S MANUAL
SOFTECH, Microsystems, San Diego, California.
Third printing Feb. 1980.

Wha 83 Wharton, Michael R.
A Note on Types and Prototypes.
SIGPLAN Notices 18,12 (1983) 122..126.

Win 76b Winkler, J.F.H.
A Program Generation Language
= [BWW 76: 5..18] (in german).

Wir 71 Wirth, Niklaus
The Programming Language PASCAL
Acta Informatica 1(1971)35..63.

Wir 75a Wirth, Niklaus
An Assessment of the Programming Language Pascal
SIGPLAN Notices 10,6(1975)23..30.

Wir 76a Wirth, N.
Comment on A Note on Dynamic Arrays in PASCAL
SIGPLAN Notices 11,1(1976)37..38.

Wir 78 Wirth, Niklaus
MODULA-2
Eidgenössische Technische Hochschule Zürich, Institut für Informatik. Bericht Nr. 27, Dezember 1978.

Wir 80 Wirth, Niklaus
MODULA-2
Eidgenössische Technische Hochschule Zürich, Institut für Informatik. Bericht Nr. 36, March 1980.

# SIGPLAN NOTICES

A Monthly Publication of the Special Interest Group on Programming Languages

VOLUME 19 NUMBER 7 JULY 1984

**Contents:**