SOME COMMENTS ON THE FORTHCOMING "EXTENDED PASCAL" STANDARD

Philippe Ranger
6120 Hutchison, Montreal h2v 4c2, Canada
20 August 1987

A note on the debate

Courtesy rules that rejected opinions be left unmentioned, save in private conversation. This certainly helps peaceful debate, but at the cost of obscurity once the debate is over. It's more often in rejecting a proposal than in accepting one that we're impelled to state the general principles we pursue -- often it seems a proposal is more likely to become the object of consensus if the reasons for proposing it are left out of the debate. Once the consensus is reached, however, those outside the deciding circle are met with the proverbial camel because all the details are clear but their common aim, if any, remains unstated.

Pascal, originally the work of one man, appeared in a context where it was quite clear what the author was rejecting: the Swiss-army-knife approach guiding what became Algol 68. I suspect that this clarity of purpose, from controversy, was one reason for the exceptional success of Pascal in a field of several other interesting small Algol-type languages.

Another argument for stretching the rules of courtesy is that, over the last twenty years, we have been forced to re-veto the square wheel (silently) any number of times. Every few years, we achieve peace at the cost of letting the next few years proceed in the same confusion that we have just worked so hard to overcome.

David A. Joslin's "highly probable" preview of the forthcoming revision of the Pascal standard [1] is admirably informative. But, in the spirit of the above, I must say that there is much missing in it. Arguments for the possible extensions are only implicit in the illustrations. Propositions rejected are completely unmentioned. And, since the aims pursued are not explicit, the quite sensible reasons not to consider contrary proposals will be clear only to those who are not proponents of these proposals.

This is not criticize Joslin's paper, which is only a preview (of a draft which, at the time of writing, we have not yet seen). But I would wish, when the draft proposal appears, that its authors be ready do explain their choices in published debate, and to answer other proposals with something more explicit than silence.

Is there a Pascal style in languages?

I believe there is, and that it follows from these hypotheses:

1. A teaching language for procedural programming can be fairly complete while remaining small, by being extremely logical, clear and simple-minded, and by being as orthogonal as possible; if this be, the language will be excellent for its purpose.

2. This should also insure a high degree of portability through machine independence.

The success of the original Pascal showed these hypotheses to be true, at least in the case in point. We may take them as guiding principles when discussing extensions. Following this success, a third principle seems also to have been confirmed:

3. With very few extensions, in the spirit of 1, the language may become excellent not only for teaching but for most of the applications of a procedural language.

And it is easy to believe that Wirth also had in mind:

4. With some care, it should be possible to specify a language satisfying 1 for which it is also fairly easy to write compilers.

4 was certainly essential to Pascal's success, and we too should keep that "impure" thought in mind. However, it speaks against 2. In fact, Pascal has some implicit machine references that are as hard to ignore as a pebble in a shoe: packed arrays, the read and write (punchcard) "procedures", missing I/O for enumerated types, file pointers, type limitations on function results, etc. Also, 4 suggests by limiting orthogonality, and Pascal has some examples of this, too: no constants for structured types, limitations on CASE selectors, strict ordering of declarations, etc.

A negative remark

Pascal not being PL/1, I cannot understand how extensions to it can be proposed without explicit regard for some principle such as above. There is no other way to tell which extended language can be called a Pascal and which cannot. My suggestions are 1 and 2, plus upward compatibility with now-standard Pascal. It is a major virtue of Pascal that the user achieves more varied and more involved applications not by learning more "language law" but by using known forms differently. Orthogonality with the core syntax must remain a criterion for "Pascalian" extensions.

There is much in the extensions foreseen by Joslin that is praiseworthy -- in fact, much that it is hard to wait for. To name a few decisions that yet will not receive unanimous praise: loop-exit, return, halt, date and time, environmental enquiries, and above all modules and separate compilation. My thought on the last is that no committee could do better than a complete crib of Modula-2, and that Joslin's prediction comes close to that, though VALUE is a restriction on Modula's module-local executable block.

What worries me, though, is that several points breach principle 1. The breaches may not be major, but they are varied enough to indicate that this principle is not a clear aim of the proposed standard. The first words of principle 1 are "A teaching language", and a consequence of this orientation is that syntax should be understandable without recourse to: "Well, that's the way it is."

The syntax of the BINDing operator and the semantics of UNBIND are sui generis, non-orthogonal with the rest of the language. So is the use of = to declare a function result variable (the same terseness could be achieved with a second pair of parentheses). Worse still is the perverse syntax introduced simply to achieve the inverse of ORD -- is it mortal sin to follow Turbo Pascal and use any ordinal type identifier as a conversion function? Also regrettable are the use of the period and the question mark in (the undoubtedly essential) type inquiries. For improvements on the last and other anti-pedagogic non-orthogonalities, see [2].

To tell the truth, I was hoping that the question mark, as well as several other non-alphabetic characters, would be allowed in identifiers, both to improve expressiveness and to facilitate the use of code-processing filters. We will return later to the question of type inquiries.

One last example. The proposed STRING type and operations are essential extensions. But, as Pascal incompletely defines the ordering of the CHAR type, all string comparisons will be implementation-dependent. In this context, the use of the extant relational operators (=, ⩽, etc.) for strings padded with spaces, and the addition a whole spate of new operators for straight string comparisons, is worse than baroque, rococo. If we are going in for single-purpose lexical categories, let's have F=, F⩽, G=, G⩽, etc. for comparisons following the French and the German alphabets! This can't be Pascal.

So as not to restrict my clarifying negativism to what appears first of all an excellent proposal, let me mention an idea which often crops up elsewhere: using line indents (overridably) in place of the wordy and distracting BEGIN and END (e.g., [3]). This does away with two major virtues of Pascal. One, total formatting freedom (now, I would have to worry whether my layout implies any block limiters or not; and any deliberate, automatic or accidental change in format could change the program at the next compile). Two, avoidance of default semantics, which is not only a condition for orthogonality but to my mind one of the most forward-looking features of the language (default semantics freeze a context of use into the language definition). No, the solution to the BEGIN - END awkwardness lies Modula-way.

From the preceding remarks, as well as those at the beginning of this letter, several positive suggestions can be drawn. Let me add two more that require some words of explanation.

The FOR loop

It is hard to have a neutral opinion on the FOR loop in standard Pascal. It's either special-purpose baroque, or an eminent aid to learning and plain clarity. I am sensitive to the first opinion, but experience as taught me the second. Hence I would like the construct's possibilities extended. This Joslin suggests in the form of FOR i IN [set]. However, a set is not congruent with the semantics of the FOR loop. The order of execution would have to follow from the order of the elements, which is meaningless for a true set. What we have here is rather an enumerated type; instead of the brackets of a set constant we should have the parentheses of an enumeration, and IN (a boolean operator elsewhere) should never have replaced the original := :

```
FOR v := (val1, val2, val3) DO
```

I do say enumerated type. The bizarre possibility in Pascal of using a control variable outside the loop, or of assigning it a value inside it, comes from using a normal variable for a loop counter. Pascal should call a loop counter a loop counter, not a "control variable". It should have no meaning, and no declaration, outside its loop. Inside the loop, it should stand as a constant, unassignable to. Hence the FOR instruction should act as the counter declaration -- with implied or explicit type.

The special rules for the "control variable" in the present standard make it downward-compatible with this proposal, if the latter also keeps the TO and DOWNTO identifiers and their semantics.

A related possible extension would be enumerations of non-ordinal-type elements: records, arrays, etc., even files. In the present standard, an "enumerated type" is half type (compatibility-wise) and half serial declaration of integer (cardinal) constants. Since, according to Joslin, we are to have structured value constructors (structured literal constants), we should of course have structured symbolic (named) constants. Thence to structured enumerations is a tempting step to take. The enumerated values would be named by symbolic constants, as always, and the corresponding variables (or loop counters) could be evaluated through the normal comparison operators, and modified by SUCC, PRED, FOR and :=. They would also be compatible with the base type, except when on the left side of an assignment. From an implementation point of view, this is simply a constant array with an implicit index. Syntaxically, it is to structured constants what the present enumeration is to integer (or rather cardinal) constants. But semantically it is a new structure: an ordered sequence.

The CASE construct

Another embarrassment is the CASE construct. As it stands in standard Pascal, it is an ugly growth of special-use syntax. There will be no debate that the extensions suggested by Joslin (OTHERWISE clause and ranges in labels) bring in a needed helping of logic. But we need not only to add to the syntax of the CASE construct, but to remove some of its specialness. If there were no OF following the selector, we could re-instate the boolean syntax of IF statements:

```
CASE ch
   = '.' : ... ;
   (> 'z') AND (<> '|') : ... ;
   IN ['?', '*', '1'..'5'] : ...
   OTHERWISE ...
   END;
```

What I like about this solution is that it is very Pascalian: we remove a syntaxic category (case labels) and find both that the semantics is clearer (cascading else if) and that the syntax is far more powerful (any type at all for the case selector). Yet, it does not forbid the present standard OF and its semantics.

Conformant constructed types

RECORD, ARRAY, SET, POINTER and FILE are not types in Pascal, but type constructors. What's awkward is that some operations are defined for all types constructed with a given constructor but, contrary to Pascal's "build it yourself" style, no further generic operations can be built from these. The language definition invokes tools for standard operators, functions or procedures that it refuses to the user's own definitions. An example of the way things should be is the string encoding-decoding proposal in Jos-

lin, which returns to the user the string coding now reserved to I/O operations.

I believe that this restriction is original Pascal's worst shortcut to compiler simplicity. Twenty years later, it has no remaining hint of an excuse. Most of the restriction could be removed by allowing parameter declarations to specify only a constructor, as proposed in Joslin, and allowing local constant declarations of the form:

```
val1: MAG construct;
val2: SIZEOF construct;
val3: MAG construct [subconstruct];
```

where MAG (magnitude) is the number of elements in construct, and sizeof the number of bytes occupied by it. Val3 shows the syntax for nested constructs. The reserved word ELEMENT could be used for files. The actual values could of course be passed on the stack at time the procedure or function is called.

The point is to have a general solution available for any constructor. It is an error to create special categories for conformant-this-or-that.

Applied to standard Pascal beyond conformant arrays, all this seems principle for principle's sake. However, the principle makes a major difference as soon as we include a string constructor, as proposed. The need for tools to build generic string functions is obvious.

Moreover, there is no reason to restrict strings (lists of simple elements) to characters. Strings should be declared using the same syntax as arrays. (This fruitful simplification, however, might have to be sacrificed to established usage.) Standard string functions should be as generic as those of arrays. With "SIZEOF element", a generic user-defined string procedure could apply to strings of integers, chars, reals, etc. Of course, implementations would limit strings to a certain magnitude.

Closing details

The general principles I wanted to illustrate are certainly clear by now. Several other points could still be made about the forthcoming standard. For instance, orthogonality would suggest that any form specified for structured literal constants also be available for READs and WRITEs of structured values. The use of enumerated types would be clearer if SUCC and PRED accepted a second parameter for non-unity increments or decrements. Nicer still would be ROLL (enum, n), increasing enum by (positive or negative) n, modulo the cardinality of the type -- in fact ROLL should apply to any ordinal type, including INTEGER.

But the one last point I would feel remiss not to mention, though it may be quixotic, concerns the file pointer. In the abstract, the standard file type constructor is as defendable as any other. However, it was never put in Pascal for abstract reasons, but for the most concrete purpose in computing: I/O. The notion of a file pointer leads to problems even with simple console I/O. The standard file type constructor maps buffered I/O only. By 1975, Wirth [4, which see] was opting for the abandonment of GET and PUT. But by then they were a basic trick in everybody's I/O coding. Eight years later, when Turbo Pascal came out with neither GET nor PUT, it was simply driven from the fold (and into the wilderness of popular success).

The habit of file-window-watching leads to disregarding another trick that is far more useful with today's large memories: massive, unbuffered block reads and writes. In Joslin's digest, I see no mention of file accesses in multiple-element blocks. But I do see that the file pointer, instead of being relegated to the status of a footbridge for upward-compatibility, remains a basic semantic element, as shown by the needlessly involved definitions for direct-access I/O. The file pointer will be confirmed, it seems, as the most dated, device-mirroring, major element in standard Pascal.

[1] David A. Joslin: "Extended Pascal -- Illustrative Features", SIGPLAN Notices, Dec. 1986.

[2] Ronald T. House: "Thoughts on 'Extended Pascal -- Illustrative Examples'", SIGPLAN Notices, Aug. 1987.

[3] Moreshwar R. Bhujade: "Visual Specification of Blocks in Programming Languages, SIGPLAN Notices, Aug. 1987.

[4] Niklaus Wirth: "An Assessment of the Programming Language Pascal", Proc. of the Int. Conf. on Reliable Software, IEEE, 1975.