COMMENTS ON A DRAFT
PASCAL STANDARD

by

Jorgen Steensgaard-Madsen

TR 79-369

Department of Computer Science
Cornell University
Ithaca, New York 14853
Comments on a Draft Pascal Standard

Jorgen Steensgaard-Madsen

Introduction

A draft for a Pascal standard to be proposed to the British Standard Institute (BSI) has been circulated by Tony Adyman heading the proposing committee. The draft is welcome as an attempt to clarify the definition of Pascal. It is a carefully prepared document forming a sound basis for a proper standard. Some polishing is needed as is usual with proposals and I see the circulation of the draft as a solicitation for comments to aid such a fine-tuning. Thus, technical comments should be read as constructive criticism pointing out areas that in my opinion need polishing.

In addition to purely technical comments you will find some pure value judgements in this document. These have their counterpart in the proposal also. However, there they appear implicit as particular choices made from an unspecified set of alternatives.

My judgements are influenced by experience gained from implementing a Pascal compiler and by drawing up an independent proposal for extension and clarification of Pascal (Steensgaard-Madsen, 1978) with emphasis on extensions, how-
ever.

The remaining part of this document contains: a section mostly based on value judgements with rather general comments, a section elaborating on a topic treated vaguely in the proposal, and finally a section with very short technical oriented comments.

The number of comments is fairly large and I consequently feel it necessary once again to point out that I support the effort to define a standard and consider the proposal an adequate basis for a final standard.

General Comments

A fundamental issue any programming language standardizer has to decide is to what extent existing implementations will be allowed to influence a standard. Ideally no influence should be allowed, but that is as utopian as having a standard prior to any implementation.

Now, implementations do exist and they differ in their solutions to various problems. Taking into account this and the facts about where Pascal compilers originate and are maintained, I think that the best a standardizer can induce is minor compiler changes and only so if the resulting compiler will satisfy the pride of the implementor in having produced a standard-adhering compiler. Further, I feel that nothing is more likely to upset and disturb an implementor than have him (m/f) admit that a certain kind of error may
occur and pass undetected. Thus, I would be very reluctant to classify any behavior as erroneous if its detection would decrease efficiency substantially and it could be traced down to clearly described cases that might be characterized as bad tricks or be ascribed to an unfortunate language definition. Instead, I would explicitly allow for variations between implementations and leave to the programmers responsibility to avoid such cases. The underlying fundamental problems should be dealt with when a language is designed, not when it is standardized.

Relating directly to the proposal I find it bad that it tries to solve such problems by allowing errors to pass undetected. This blurs the distinction between the concepts "error" and "implementation dependent". A modification of the proposal in this respect should not merely sharpen the handling of errors. Some situations now characterized as errors should be allowed to lead to implementation dependent behavior however deplorable this may be.

As an aid for programmers as well as implementors it would be very nice if the standard would provide a form to be completed with information on implementation dependencies (directly in the form or via a reference and including errors in the sense of the proposal).

The terminology of the proposal is unfortunate in a few cases. First of all it is confusing to claim that the proposal defines "Standard Pascal". The language defined is
not identical to the language Standard Pascal described by
Jensen and Wirth (1975). More appropriate is BSI Pascal or
ISO Pascal. Secondly, the proposal should not neglect ex-
isting tutorial literature on Pascal by the use of the terms
"ordinal type" and "case-constant" when the terms "scalar
type" and "case-label" are well established.

Turning to more technical points I find the introduc-
tion of a particular undefined value unfortunate. Few com-
puters have hardware for the support of the proposed associ-
ated requirements (cf. sections 6.5.3, 6.6.4.1.2, and 6.7)
and the costs of software support are generally prohibitive.
(See also comment 1. in the Details section).

The requirements proposed for the use of the record
variants are not enforceable with reasonable costs. The
worst complications arise from the use of tag-fields and
fields of variant parts as variable parameters with the fol-
lowing prospects:

(a) the assignment to variable parameters will depend upon
the actual parameter being a tag-field or not

(b) the reference to a variable parameter may become er-
roneous because the actual parameter is a field of a
variant that ceases to exist due to a change in a tag
field value

(c) the interpretation of the assignment to variable param-
eters is complicated further by the possibility to pass
in any one call actual parameters R.A. and R.B., where
A and B are field identifiers of different variants.

The concept of a record type with variants in Pascal allows
for unsafe constructs. No doubt Niklaus Wirth was aware of
this when he designed the language. The proposal may even
reflect what was originally seen as misuse of the concept.
But this does not mean that misuse should lead to an error.

The rule for accessing fields of variants in a record
without explicit tag-field requires a 'hidden' tag field to
be included. This may not be desirable as illustrated by
the following example

    case kind: integer of
      1,2: (a: real);
      case (kind:) integer of
        1: (b: real);
        2: (c: integer) )
      3: ...

In this way one may in perfect control distinguish conceptual
variants (a,b) and (a,c) with a common field identifier
"a", which is not otherwise allowed.

The proposal allows a forward declaration of a pro-
cedure or function to be followed by a proper declaration.
However, it is required that the parameter list and the
function result type be omitted in the proper declaration.
I find this inappropriate and in bad style. Anyone using
the text of the proper declaration will probably first look
to its heading to provide information on identifiers not de-
finied in the block. Now, except for function headings being
syntactically distinct for the situation anyone is likely to
be misled. It is worth noting, that the mark of distinction
for function headings is its non-conformity with the pro-
posed syntax.

Previously, implementors have been allowed to let
dispose be an operation without any effect. This has been
pleasing because it is absolutely safe. Again, immense com-
lications arise from the proposed definition of dispose as
a concealed multiple assignment to a dynamically determined
set of variables. Further, the description of errors does
not cover the situation where a field of a record is in use
as a variable parameter when the record is disposed of. Fi-
nally, you may want to contrast the errors with the situa-
tion

\[ p.a := E \]

where \( E \) may cause a dispose\((p)\). The latter is implementa-
tion dependent!

An Omission

It is important to me also to point out at least one
area in which the proposal could be more specific. Unfor-
tunately it is an area in which the variation between imple-
mentations is great and of real importance for many programs. Of course I am thinking of input of legible data.

The following problems are important

1. What encoding of characters is used?

2. Where does a legible line terminate?

3. What is the initial status of the predefined text variable input?

Character encoding has always been a problem that has an obvious solution. However, it has been customary to give in to efficiency considerations and settle for a machine dependent encoding. I have little hope that this attitude will change and mention the problem for completeness only.

The division of a text into lines may or may not correspond to the physical appearance of a newline character. The Pascal operations "eoln" and "realdn" reflect this very neatly. However, reading composite items like integers and reals is affected by the details of the division. Here is a list of some approaches.

Pascal 8000: lines usually contain a fixed number of characters.

UCSD Pascal: lines may be of any length.

Pascal 6000: lines may be of any length, but the number of characters is an integral multiple of a fixed number.

Pascal 1100: a line may hold a limited number of characters
but never ends with a space character.

Now, reading a simple sequence of integers from input cannot be expressed simply like

{erroneous scheme}
while not eof(input) do begin
  read(i); P(i)
end

Since the proposed structure of text is ![character] line marker] the last integer is not terminated with eof(input) = true. Presumably the proposal requires the above scheme to terminate in error ("reading ceases as soon as [...] input contains a character which does not form part of a signed-integer"). Many implementations will process the last integer twice and not report any error.

Learning from the scheme for processing characters one at a time:

while not eof(input) do begin
  while not eoln(input) do begin
    read(ch); S(ch)
  end;
  readln
end

we recognize a need for an inner loop to process all numbers in a line. The terminating condition is, however, hard to state nicely taking the various approaches into account. It is easily stated in ordinary language: the inner loop must terminate if only blanks remain in the current line.
An easy solution to the problem of stating the desired terminating condition is to introduce a new predefined function for this purpose alone. In order to tighten the definition as much as possible the function could be defined thus:

\[
\text{function eol(var f: text): boolean; begin}
\text{if not eof(f) then}
\text{while (f = ' ') and not eoln(f) do get (f);}
\text{eol := eoln(f)}
\text{end}
\]

The price paid to satisfy all implementations is to accept a predefined function with side-effect. It can be done without side-effect but only with the inclusion of a hidden, limited buffer for look-ahead over blanks. Note that the Pascal IIIIC approach allows a simple terminating condition: eoln(input).

Corresponding to the eol predicate we have the following to test whether more information remains in a textfile:

\[
\text{function eot(var f: text): boolean; begin}
\text{while eol(f) do readln(f);}
\text{eot := eof(f)}
\text{end}
\]

(The simple condition is based on the proposal that eof(f) = true implies eoln(f) = false. However, it is stated rather vaguely and I may have misinterpreted the proposal).

Now it is possible, with the latter predicate, to read
and process a sequence of integers in a simple manner:

    while not eot(input) do begin
        read(1); P(i)
    end

For writing programs interacting with a human at a terminal the primitives of Pascal and the predicates above are adequate only if the unit of exchange is a line. The UCSD Pascal system, for one, is not based on linewise interaction, and I don't know whether just a few additional primitives may be defined to satisfy that approach. Presently, the problem is resolved by a predefined type similar to text.

It is not true, however, that Pascal in general is unsuitable for writing interactive programs if line-oriented interactions are acceptable. If so it is reasonable to read one line into a hidden buffer by every readln. (Consequently, some proposedly interactive programs should apply read where readln is applied.)

The initial status of input is important for the behavior of interactive programs. If the first character written by a human operator must be available to the program when it takes control it is impossible to give an initial "prompt". The problem is serious enough to deserve an elaborate explanation of one solution that has worked well for a number of years in the Pascal 1100 compiler.
a. The structure of input follows the general pattern proposed for text variables, but is more specifically described by

\[ \text{linemarker)((character) linemarker)} \]

i.e. eoln(input) is always true when processing is initiated. The first "linemarker" is generated automatically to precede whatever is written by a user.

b. An explicit reset(input) is interpreted as readln if no other operation on input has been performed.

I strongly recommend that the initial status of input be standardized in this way. Programs that initially read a number will not be affected at all, and a simple cure can be prescribed for any affected program: include an initial reset(input) operation.

In summary, I recommend

1. Automatic suppression of trailing blanks in every line

or

inclusion of predicates like eol and eot

2. eoln(input) = true when processing starts.
Details

This section consists of a list of objections to the proposal mainly on a level just above correction of typing errors.

a. It is not clear what is meant by: "... there need not be a definition ..." (section 3. on implementation dependent)

b. Directives should be mentioned as lexical tokens (6.1)

c. Scope is defined in terms of the undefined and unfamiliar notion of a range (6.2.1)

d. Section 6.2.1 is generally imprecise ("the occurrence" in (2), "corresponding occurrences" in (5), "the domain of a pointertype" in (5); parameters ought to be mentioned)

e. The rule by which T1 and T2 are identical if T1 = T3 and T2 = T3 ought to be more explicit

f. Are parameter-identifiers not allowed as entire variables (6.5.1)?

g. The set of directives should be implementation defined rather than implementation dependent (6.6.1)

h. A type-identifier rather than type must be required in a parameter group (6.6.3)
position with respect to deallocation and file buffer
change

R. The example "transpose (a,n,m)" is deceptive (6.8.1.2)

a. Use case-index or selector consistently (6.8.2.2)

u. An example of how to read a sequence of integers from
   input with intelligent use of eof will be welcome (6.9.1.3)

V. Does page(f) imply a writeln(f) (6.9.5)?
   (de facto word symbols, 6.11)

Conclusion

The draft of a standard proposal for Pascal comes as a
promise of a future cleanup in the description of Pascal.
Although I do not take the same points of view as the pro-
posing committee, I wish for success for the effort in gen-
eral.

The main points of this paper can be summarized as fol-
loows:

a) the proposal takes a too strict (and in certain cases
   inconsistent) position in defining what should be con-
   sidered erroneous behavior.
6.6.1.1 seems not to be consistent with the article 6.6.17.4, the proposition for implementation dependencies in parallel programs should only be defined for strings of characters.

6.6.1.2: compute a non-negative value (6.6.4.2.1)

6.6.1.3: every call to is not stated explicitly in 6.6.9.1.1.

6.6.1.4: this is the reason why the compiler allows the use of variables in a for statement.

6.6.1.5: the formation of a proposition is a break of style.
Cognitex & Kenzer.

Recommended Extensions, DIXY, Sturgeonhead 81, DX-2200


For Science. University of Manchester, Oxford Road, Manchester.


References

Respect to parameters that are excellent and long overdue.

In contrast to comprehensive types, food and the extraction with

find the proposals are well organized, the dissemination of

on the positive side, which is too easily forgotten.

for implementation.

from the common concern for the deprecation of parameters.

above on the organization where everyone would benefit

the draft needs polishing of a number of details.

(0) basic problems concerning input are not solved.

(0) does not support good programming style.

The rules for composition forward and proper deprecation

- 15 -