SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 123-130 (1981)

Forward-declared Procedures, Parameter-lists and Scope

A. H. J. SALE
Department of Information Science, University of Tasmania, Hobart, Tasmania

SUMMARY

This paper examines some issues which relate to the definition of scope in Pascal, and pressures which have been brought to bear on the draft international standard for Pascal for duplicate procedure headings for forward-declared procedures. The causes for these pressures, and the conceptual integrity of Pascal are discussed. The interfacing nature of parameter lists is examined, and the concept of 'detergent construct' introduced, leading to 'detergent scope rules'.

KEY WORDS Forward-declarations Parameters Scope Pascal

INTRODUCTION

In the processing of the sequence of working drafts (ISO/TC97/SC5 documents N462 and N510) of an International Standard for Pascal,[1, 2] many comments were received suggesting that the definition of Pascal should be changed so as to require a formal parameter list in the second part of a procedure which has been declared 'forward'. The original definition, and the working drafts, require that the definition of the block of a procedure declared 'forward' does *not* have a parameter list following the procedure name. (In this paper, **procedure** is used generically to denote either a Pascal procedure or a Pascal function, with the obvious small differences that exist.)

These comments have not been sufficiently convincing as to result in a change to the language defined by the working drafts. The reasons for this are both interesting and important, and illuminate some preconceptions with which computer languages are often viewed.

THE CRITICAL COMMENTS

To understand the issues, it is necessary to understand the critics of the Pascal definition. Their comments hinge around the provision in the language for separating

0038–0644/81/020123–08$01.00

Received 22 January 1980

the specification of a procedure-heading and its block-definition by the 'forward declaration' feature, and are illustrated by the example in Figure 1.

```
procedure Handle (this : Thing; what : Treatment);
  forward;
...
{other procedure declarations}
...
procedure Handle;
  ...
  {block of Handle}
  ...
end;
```

Figure 1. Example of forward-declared procedure

The critics of this particular syntax point out that the second part of the procedure is often separated from the first part by many lines of text, if not pages. Consequently, it is argued, the procedure-heading information is not readily accessible when reading the text which constitutes the block. It is therefore suggested that the procedure-heading information be repeated in the second part of the procedure and that the compiler be required to check that the two sets of information are identical. In this context, *procedure-heading information* means the formal parameter list (if any), and the function-type (if applicable).

THE NAIVE COUNTER-COMMENT

The usual reaction of Pascal-users to this suggestion is to point out that there is an alternative way of achieving the objective. If the parameter-list is indeed so important that it need be put in the second part, then it can be included there within comment delimiters. The same example is shown in this style in Figure 2.

```
procedure Handle (this : Thing; what : Treatment);
  forward;
...
{other procedure declarations}
...
procedure Handle { (this : Thing; what : Treatment) };
  ...
  { block Handle }
  ...
end;
```

Figure 2. Example with commented supplement

Such a response is correct, but is not sufficient to completely dispose of the criticism. It concedes the point and proposes a palliative. Nothing enforces the existence of the comment, except perhaps some in-house programming standard which programmers are exhorted to follow. Even worse, there is no guarantee it is right, and a few generations of maintenance may well make it quite probable that it is not, as changes to the program cause the first heading to be altered and the second forgotten. It is, after all, quite likely to be far away from the point of definition of the heading.

THE IMPLEMENTOR'S CRITICISM

A more direct criticism of the repetition proposal is that raised by implementors of Pascal. The main issues are the size of the checking code and the definition of the check to be performed.

Size

If the language requires two instances of a parameter-list, then a compiler that does not check that the two are 'identical' is irresponsible. Such checking must add to the compiler's size, but gives no increase in expressive power. The evidence indicates that the size increment is appreciable, though naturally small in comparison to the entire compiler.

Definition

One possibility is to insist on textual-identity: the two pieces of program text are character-for-character identical, presumably also embracing line transitions. As long as no redefinitions of identifiers in the list is possible (as would be sensible), this would guarantee correctness. However, an exact textual match would sit uneasily in Pascal, since there is no other corresponding feature with this property. It would be likely that token-equivalence (at the very least) would be considered preferable, so that the exact number and type of token separators need not match (spaces, newlines, comments). But this too is without precedent in Pascal, and implies that the accompanying commentary may not match—which is exactly the problem we started out with. Yet a third proposal is to define some sort of 'structural equivalence'. This runs into several new problems to be resolved; for instance which of the pairs of examples in Figure 3 are to be considered 'identical'?

```
type  angle = real; radius = real;
      digits = 0..9; decimal = 0..9;

      (a,b : real)
      (a : real; b : real)

      (a : angle)
      (a : radius)

      (d : digits)
      (d : decimal)

      (dummy : integer)
      (count : integer)

      (procedure p (x : real))
      (procedure p (y : real))

      (x : real) : angle
      (x : angle) : real
```

Figure 3. Similar procedure heading fragments

All these questions can be resolved *if necessary*, and some of them have had to be tackled in order to define the related concept of parameter list congruity in N510. However, what such rules do is to add to the complexity of the language, with implications for teachers, learners and users, for implementors, and for standards-definitions. It can be expected that this increase in complexity will have an effect also on the general standard of correctness of compilers, especially as the checking is purely a check and has no inherent semantics.

Thus from the implementor's point of view, repetition of the procedure heading information is simply a pain-in-the-neck. This view is shared by standards-writers. The benefits for users also appear at least debatable, in view of the points raised above.

THE ABSTRACT COUNTER-ARGUMENT

All the above problems with the implementation of the repetition suggestion should indicate to the interested reader that there is something conceptually wrong. And there is: the danger signals are there because it violates a basic principle—that an object of a programming language have *one and only one* definition. The suggestion modifies the language to require two definitions which must then be checked to be 'identical', and quite naturally this leads to all the familiar problems of *a posteriori* structural incompatibility. Largely due to careful design and sympathetic standardization, Pascal has avoided most of the structural incompatibility mess and the few examples are recognizably due to late modifications (congruity, some aspects of conformant array parameters, the so-called string-types).

The manifestation of the problem here is due to two major causes. Both set up mental blocks so that the obvious solution cannot be seen, and they deserve individual attention.

Programs as structured objects

There is a tendency to think of a program as a sequence of tokens. Of course, this is how language designers lay out the syntax and how standards specifications are written. But a program is something much deeper than that—it is a structured object.

To illustrate, conceive of the example procedure given earlier as being embedded in an entire program. At the level that it is defined, the only attributes of the procedure that are relevant to the embedding matrix of text are the procedure name, the type(s) of the parameters and whether they are variable or value parameters, and the function-type (if relevant). Thus the heading should seem as though viewed through virtual red-and-green 3-d spectacles, so some parts become indistinct and fuzzy:

procedure *Handle* (**** : *Thing*; **** : *Treatment*);

Ignoring a single complication (the possible existence of non-local gotos), the rest of the procedure text is invisible and irrelevant, as though hiding behind the heading which is its external interface.

If we seek to find out about the procedure, then we shift the focus and colour of our virtual spectacles, and we can view the identifiers that denote the formal parameters, the definitions and statements of the block, and the interfaces of any further refinements. This is closer to a realistic model of a program, and if this is adopted as the model then the picture of forward-declared procedures changes dramatically!

All of a sudden we see that the view of 'normal procedures' is perhaps not so obvious as had been thought. The heading of a procedure and its block (in Pascal terms) are two distinct things, which distinctness is preserved by the forward-declared procedure form. The 'normal' form in which the procedure heading and its block are contiguous can be seen to be a convenient short-hand notation; syntactic sugar if you wish. The process of squashing the structured object called a program into the linearized and flattened form we know as Pascal programs is what creates the conflicts we saw earlier. Or do we still see them as conflicts?

Conventional Pascal style suggests that procedures are not declared forward unless this is essential, and then when a mutually recursive situation does arise only the fewest number of such forward declarations are made. This should now be considered to be a bad practice, for if two or more procedures are mutually recursive then declaring them *all* as forward procedures in the same place (with appropriate commentary) clearly indicates their mutual interdependence. That this is important can be easily understood when it is realized that the correctness of a group of mutually recursive procedures depends on all the members of the group.

The other insight is that a procedure should have only one heading, wherever we have to put its block so that it can be compiled conveniently. If two headings have to be introduced by the flattening process, we should do this only with the utmost reluctance.

Programs and listings

In stark clarity, we can now see the second flaw in the argument for repetition. It was obviously assumed by the critics that a program and a listing are equivalent, for they were arguing for a change to the syntax of Pascal while pointing to problems with reading a listing (or screen). But programs are *not* identical to listings. A Pascal program is a representation of an abstract program we have conceived, and a listing or screen display is still more a representation of a Pascal program. It is not 'the program' itself. Even the most primitive compilers produce listings that have more features than are in the program text, and more sophisticated tools for browsing through programs or producing structured printed forms are easy to conceive.

The answer to the critics is now quite clear: the notion of changing the language to have two headings is rejected because it would degrade the conceptual integrity of Pascal. If it is important to see the procedure-heading near the text it is the interface for, then it is the compiler's job (or a software tool's job) to reproduce on the listing at the appropriate point the definition it has of the heading.

Armed with this idea, we can generate a plethora of possible actions that industrious implementors might try out in the market-place:

* Print out the heading at the location of the second part of a forward-declared procedure only.
* Print out the heading at the 'begin' and 'end' that delimit the statement-part of a procedure, perhaps with the first conditional on the presence of local procedures or functions, and the second conditional on statement-part length.

* Print out at the top of each page an indented form of the headings of all enclosing procedures at that point in the text.
* Provide a software tool which documents all headings and the locations of the headings, corresponding blocks if separated, and statement-parts if separated by intervening procedure and function declarations.

Clearly there is no problem when you look at the issue in the right way. To demonstrate this, the software tool outlined in the fourth of the possible actions has been implemented.[3] It is also interesting to note that the information a compiler needs to check whatever definition of 'identity' is adopted, is *exactly* the same information needed to allow reproduction of the parameter list in the listing! Consequently the problem only exists for those who insist that *one* tool should suffice for all purposes they can conceive. In this case, programming language syntax is clearly the wrong tool.

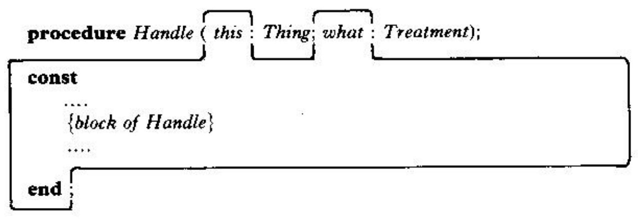
THE DETERGENT MOLECULE VIEW OF PARAMETERS

Having come this far with the examination of the original problem, it is appropriate to examine some further insights into parameter lists thrown up by Pascal. It has been argued that the procedure heading is an interface between the embedding matrix or medium, and the internal details of the procedure block. A very close analogy, which will be used to name the concept, is that of a *detergent molecule*. Such a molecule has a number of chemical chains (or sites) which have an affinity for water, and are 'soluble' in it. Other chains or sites have an affinity for oil or grease. The two kinds of tails are called hydrophilic (water-loving) and hydrophobic (water-hating).

A procedure heading is therefore a *detergent construct*. It contains parts which have an affinity for the outer medium (procedure-identifier, parameter list types, function-type, variable- or value-parameter kinds) and which are all the outer medium can see, and parts that have an affinity for the interior of the procedure (the formal parameter-identifiers).

Examining this detergent construct between us and the procedure block, it can be seen that the notion of 'scope' is bound up with the two sides of the interface. The parameter-identifiers properly belong in the scope of the procedure, for they are not known outside it. On the other hand, the outwards-facing identifiers properly belong in the enclosing scope, for the procedure is not usable unless its types and identifier are known. What this suggests is that the scope of the interior of a Pascal procedure (or one in a Pascal-like language) should be recognized as including all formal parameter-identifiers and the procedure block, but not any other component of the procedure-heading.

The only objection to this is that the portion of text over which the scope reigns is not contiguous: the parameter-list portion consists of separated regions, as indicated in Figure 4. The implementation of the scheme would not be difficult, and would simply entail controlling the apparent scope level while scanning the parameter-list. That the suggestion is reasonable can be seen by briefly examining the corresponding situation in Algol 60. In that language, all types were represented by word-symbols of totally immutable meaning over the whole program and both the procedure-identifier and the function-type (if appropriate) preceded the parameter-list. Thus a simple textual division achieved exactly the same effect that in Pascal requires a little more work.



```
procedure Handle ( this : Thing; what : Treatment);
const
    ....
    {block of Handle}
    ....
end ;
```

Figure 4. Detergent scope in Pascal

Scope is not, of course, defined this way in the current draft international standard for Pascal. For one thing, very few (if any) compilers for Pascal implement scope this way at present, and a standardizer must be very sure of his ground before invalidating all current compilers! However, scope has to be well-defined, and the definition contained in the Working Draft presented to the Turin meeting of ISO/TC97/SC5 (N510) contains an acceptable compromise which is very close to the position stated above; in fact it is slightly more restrictive in that there are a few unpleasant procedure headings which it is impossible to write under N510's rules, but would be permissible under Detergent rules.

This solution to the scope problem for parameter-lists states that there are two regions (ranges, places, text-fragments) of a program which belong to a procedure and which may have scope properties. The first is the parameter-list, and the second is the procedure-block. A definition of a formal parameter-identifier then constitutes two definitions:

1. As a parameter-identifier within the parameter-list scope, and
2. As a variable-identifier within the procedure-block scope.

Both the N510 rules outlined here, and the Detergent rules, allow for scope rules to be enforced uniformly by both one-pass and multi-pass compilers, using rather simple algorithms.[4] Pascal compilers that have inherited from Pascal-P an inability to distinguish programs with correct or incorrect scope usages can therefore be altered slightly so as to be more correct, and the concept of scope is kept as simple as possible.

A BRIEF COMPARISON WITH ADA

By way of contrast, a brief look at the structure of Ada[5] in these areas is interesting. The key features to note are:

1. The forward-declaration feature of Pascal is also in Ada. However, Ada *requires* full repetition of the procedure heading (called subprogram specification) if the subprogram-declaration and subprogram-body are separated.
2. Ada requires that the two specifications 'must be identical', but the definition of identity is elusive. The most plausible interpretation seems to be token-identity.
3. The concept of scope is present in Ada, but it is not the same concept as that of Pascal; indeed the resemblance is slight. In Pascal one can talk about the scope of the interior of a procedure, but in Ada virtually each identifier has a different 'scope'. For instance: 'The scope of a declaration given in the declarative part of a

block, subprogram body, or module body extends from (and includes) the declaration up to the end of the corresponding block, subprogram, or module.', which is one of nine scope rules.

4. There is an attempt to achieve the effects of normal scope rules by the addition of restrictions: 'An identifier used (as opposed to being declared) in a declaration (or component) list may not be redeclared in subsequent declarations of the same list.' Conceptually, therefore, the use of an Ada identifier causes it to assume an immutable state for a particular region of text—a sort of instant reserved word.
5. The scope rules for Ada's formal parameters are similar to the treatment in N510 for Pascal, as far as can be determined. The published descriptions of Ada do not give sufficient formalism to be able to determine if they are identical.
6. The formal parameter-identifiers are conceptually knowable and known to the embedding procedure since they may be used in actual procedure calls to identify the parameters. The closest analogy in Pascal is that of the field-identifiers of a record-type.

With the exception of points 5 and 6, these features are distinctly unabstract. The conclusion must be drawn that Ada's treatment of procedure headings and scope are based on the linearized view of an Ada program, and are not one of those Pascal-based features that one reads about. Insisting on a linearized, or flattened, view of a program can be likened to believing in the Flat Earth Theory . . .

ACKNOWLEDGEMENTS

This work was partially carried out in Turin, Italy while attending an ISO/TC97/SC5 meeting, and partially while on sabbatical leave at the University of Southampton, England. The arguments and co-operation of W. B. Price, A. M. Addyman and J. Miner are gratefully acknowledged.

REFERENCES

1. A. M. Addyman, 'Specification for Computer Programming Language—Pascal', *Document ISO/TC97/SC5 N462*, February 9, 1979. (Reprinted with slight differences in *Pascal News No. 14, IEEE Computer,* and *Software—Practice and Experience.*)
2. A. M. Addyman, 'Specification for the Computer Programming Language Pascal', *Document ISO/TC97/SC5 N510,* October 4, 1979.
3. A. H. J. Sale, 'The Design of a Software Tool', *Department of Information Science Technical Report,* University of Tasmania, 1980.
4. A. H. J. Sale, 'Scope and One-Pass Compilers', *Australian Computer Science Communications,* **1** No. 2, (April 1979). (Reprinted in *Pascal News No. 15*).
5. U.S. Department of Defence, 'PRELIMINARY ADA REFERENCE MANUAL' and 'Rationale for the Design of the ADA Programming Language', *SIGPLAN Notices,* **14**, No. 6, Parts A and B, (June 1979).