A Comparison of Pascal and Ada

B. A. Wichmann

Division of Numerical Analysis and Computer Science, National Physical Laboratory, Teddington, Middlesex TW11 0LW, UK

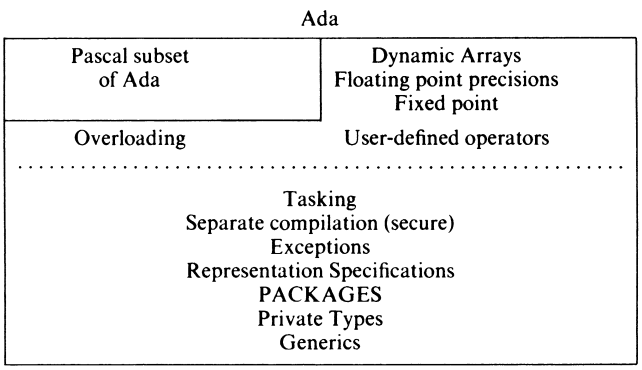
This paper compares the two programming languages, Pascal and Ada. While Ada is based upon Pascal, its design objectives are very different. Pascal was designed for teaching whereas Ada was designed for major military software systems. The simplicity of Pascal is advantageous only if its restrictions does not jeopardize the programming of an application. The improved modularity of Ada, as provided by packages, should be an important aspect for commercial development.

INTRODUCTION

The new programming language Ada is based upon Pascal.[1] It is natural, therefore, to compare them, in spite of the fact that they have been designed with quite different objectives in mind. Pascal was designed by Wirth as an educational tool.[2] The facilities it gives are just sufficient for modest undergraduate projects. The discipline of such a Spartan language is ideal in such an environment, but cannot be recommended for major commercial or industrial projects. On the other hand, Ada was designed to meet a wide range of objectives specified by the US Department of Defense,[3] which would inevitably lead to a larger and more complex language. The requirements document prepared by David Fisher was certainly ambitious, even contradictory, but at least it provided the language design teams with a clear statement of the objectives. This paper does not survey the Ada language for which the reader can consult Barnes.[4]

OVERVIEW

The Ada language is five or six times the size of Pascal. One can see this at a superficial level by counting syntactic productions, pages in the manual or number of lexical units. All the indications are that compilers will be five or six times the size of that of Pascal (given comparable code quality). At a deeper level, one can enumerate the facilities that Ada contains that have no equivalent in Pascal. This gives the diagram (drawn to scale):



Above the dotted line are features which 'could' be added to a Pascal-like language without radical revision. Below the line are facilities of Ada which have a major influence on the whole language design.

It is tempting to analyse a language in terms of a list of features alone, but this is not possible because of the interaction of the features themselves. The Ada design is a serious attempt to provide a coherent structure to a limited set of facilities that are essential for the intended application domain of the language.

One might suppose that Pascal would have a significant advantage over Ada in terms of having a stable and precise definition. Unfortunately, the Pascal report is defective in almost every detail, whilst giving adequate information for the ordinary programmer. Hence a formal standard was needed which would encompass the *de facto* definition,[5] but would specify every detail appropriate for an international standard. The work was undertaken first by BSI[6] and then by ISO, both groups being led by Dr A. M. Addyman. It soon became apparent that merely giving more substance to the Jensen and Wirth report was not, in itself, adequate. Hence a language design effort was being undertaken on Pascal at the same time as that for Ada.

HOW SIMPLE IS PASCAL?

It has been claimed by many that Pascal is a very simple language. That it is much simpler than Ada is not in question, but a glance at the ISO standard for Pascal soon shows that it is not truly simple. This lack of simplicity can more easily be illustrated by considering a number of examples:

Type equivalence

The Jensen and Wirth report did not say when two types were 'the same'. In an important paper,[7] two approaches were considered; structural equivalence, meaning that the constituents were the 'same' (to be applied recursively), or name equivalence meaning that the identical identifier is used for the name of the types. Both Ada and ISO Pascal use name equivalence. However, Pascal cannot *just* use name equivalence because a string such as 'ABC' has no type name associated with it. Hence it is structurally equivalent to a packed array [1 . . . *n*] of

CCC-0010-4620/82/0025-0248 $02.50

char (for some n). Also, to make the passing of procedures and functions as parameters secure, ISO Pascal requires that the parameter specification is given for such procedures, for example:

```
function INTEGRATE (function F (x: real): real;
                    lower, upper: real): real;
```

Any function can be passed as an actual parameter corresponding to F if it has the same parameter structure (a single real parameter by value, returning a real result).

Welsh *et al.*, consider having a language with only name equivalence but this would require more changes to Pascal than could be accepted as part of its standardization.

Array parameters of different actual sizes

The Jensen and Wirth definition of Pascal required that formal and actual array parameters were of the same type which implied that they were of the same size. Hence standard numerical computation or even a sort procedure is not possible in original Pascal. ISO Pascal introduces an enhancement (at level 1) to overcome this. It effectively introduces structural equivalence in the parameter position for formal array parameters which are appropriately declared, for example:

```
procedure MATRIXMULT (a: array [a1low ... a1high,
                               a2low ... a2high] of real;
                      b: array [b1low ... b1high,
                               b2low ... b2high] of real;
                      c: array [c1low ... c1high,
                               c2low ... c2high] of real);
```

The subscript bounds are passed over effectively as additional parameters, accessible as constants within the procedure.

This significant addition to Pascal has met with some opposition, but it provides a much wider application area for ISO level 1 Pascal, which I hope will encourage its implementation. (Procedures for string handling, similar to those of UCSD Pascal can now be written in the standard language).

Overloading

In Pascal, the function abs takes a real or integer parameter and returns a value of the same type. It is as if one had two functions:

```
function abs (x: real): real; ...
function abs (x: integer): integer; ...
```

This cannot be written in standard Pascal since an identifier cannot be declared twice in one scope. In contrast, in Ada this is permitted and mirrors the facilities with the operators such as '+' and '*', which the user can define himself.

Default parameters

Input-output procedures and functions have a single default parameter (the file 'input' or 'output' as appropriate) in order to provide a more convenient interface to the user. In contrast, Ada has a standard method for default parameters which is, in consequence, available for user-defined procedures.

Input-output statements

Pascal has 'procedures' read, readln, write and writeln for input-output, which take a variable number of parameters. Apart from the initial (possibly defaulted) file parameter, the remaining parameters are the values to be output or variables to be input. Implicit conversions to/from characters are performed for files of the special type 'text'. In contrast, Ada introduces no special language features for input-output, relying upon overloading, default parameters and packages to provide a comparable system to that of Pascal. To output an integer and real in Pascal to the default file one would write:

```
write (I, X);
```

whereas in Ada, with the fixed number of parameters one would have:

```
PUT (I); PUT (X);
```

where PUT is overloaded for the integer and real types involved.

THE EXTRA FACILITIES OF ADA

None of the additional facilities of Ada can be conveniently expressed in terms of the basic Pascal-like subset. Moreover, there is a clear need for these facilities in writing large real-time systems for which Ada was designed. For those who think that Ada is too complex, please consider how applications demanding these facilities should be programmed. The major facilities are as follows:

Arrays whose size is determined at scope entry

Algol 60 allows one to declare arrays whose size is dependent upon the data. Pascal is a backward step in this case, but an essential one if different types are to be freely composed into records and arrays. Ada provides the Algol 60 array mechanism, but has restrictions on how dynamic arrays can appear in records. Hence Ada provides the functionality of both Algol 60 and Pascal at the cost of some complexity in extreme cases.

Varying integer and floating point sizes

Pascal cannot be used seriously for much numerical work because it does not permit varying precision of floating point or different integer sizes. Ada provides this by allowing compilers to implement several sizes of integer and floating point types. Dependence upon the particular

hardware can be avoided by using the *derived* type mechanism. For example with

```
type REAL is digits 5;
type INT is range -1000 .. 1000;
```

the compiler implements REAL and INT with a hardware type which has at least the necessary precision or range. Operations on type REAL and INT are then derived from those on the hardware types. By this means, users can write truly portable Ada programs which is not possible in FORTRAN or Algol 68.

Fixed point data types

Input and output signals from sensors are essentially fixed point rather than floating point. Although fixed point is awkward to program because of the scaling that is necessary, there are a few, but important, applications for which fixed point is essential. Floating point is not adequate in such situations, because the machine may not have the necessary hardware or perhaps because of the additional space required for floating point data. The standard functions such as SIN and COS are often programmed in fixed point (in assembler) whereas these can now be done in Ada using either fixed or floating point.[8]

Exceptions

In many real-time systems it is essential for them to continue to offer a service (perhaps degraded) in spite of hardware malfunction. A telephone exchange control program or chemical plant control program would be typical. Many error situations are easily anticipated and can be handled directly at the point of detection. Others are more difficult in that the remedial action needed depends upon the circumstances which interacts with several levels of the system design. Exceptions in Ada allow one to program for these circumstances without the error handling code submerging the straightforward case. An *exception* can be raised at one level and then handled at a higher level, depending upon the calling sequence of the events which led to the exception. Ada also *defines* in a way that even ISO Pascal does not, which errors must be trapped by a compiler and those which give rise to a run-time error (i.e. an exception).

Tasking

The addition of concurrency facilities to languages like Pascal is a growth area for both academic study and serious exploitation (not incompatible!). Ada continues this tradition. The rendezvous mechanism for mutual exclusion in Ada is elegant but largely untried. Of all the extensions in Ada, this is the most ambitious and yet the one which has met with the least opposition. Only time will tell how good the particular design is.

Packages

All large Pascal programs suffer from an essential defect: they lack any clear module structure (i.e. between a procedure and a program). For instance, a compiler is naturally broken down into lexical, syntactic, semantic and code generation phases. Pascal compilers (almost always written in Pascal) only show this structure by means of comments. In practical terms, packages are probably the most important improvement of Ada over Pascal because they allow for the effective exploitation of program components. A package can consist of two parts: a specification and a body (which implements the specification). Typically, the specification and the body will be compiled separately—the first during program design and the latter during the (longer) implementation phase. Code using a package can be compiled as soon as the specification has been compiled, so that the language naturally supports top-down program development.

Apart from providing the natural unit for separate compilation, packages are also the unit for the implementation of abstract data types. Pascal, for instance, has the concept of a file built into the language. This is not necessary in Ada because files can be (and are in the standard input-output package), implemented as abstract data types. This is done as follows:

```
package I_O_PACKAGE is
   type FILE is limited private;
   procedure OPEN (F: in out FILE);
   procedure CLOSE (F: in out FILE);
   procedure READ (F: in FILE; ITEM: out
INTEGER);
   procedure WRITE (F: in FILE; ITEM: in
INTEGER);
private
   type FILE is
        record
        NAME: INTEGER: = 0;
        end record;
end I_O_PACKAGE;
```

The part after 'private' contains hidden details of the abstract type FILE so that effective compilation of users of this package is possible. (Users can declare objects of type FILE, hence the space and alignment rules for these objects must be known to the compiler). Note that the specification is quite small since it only contains the interface between the users and the implementation. In contrast, the package body is much larger, which in outline might be:

```
package body I_O_PACKAGE is
   LIMIT: constant: = 200;
   type FILE DESCRIPTOR is record . . . end record;
   DIRECTORY: array (1 .. LIMIT) of FILE
   DESCRIPTOR;
   . . .
   procedure OPEN (F: in out FILE) is
             begin . . . end;
   procedure CLOSE (F: in out FILE) is
             begin . . . end;
   procedure READ (F: in FILE; ITEM: out INTEGER)
             is
             begin . . . end;
   procedure WRITE (F: in FILE; ITEM: in INTEGER)
             is
             begin . . . end;
begin
   . . .
end I_O_PACKAGE;
```

The procedure bodies for OPEN etc are not provided, the specification of them being repeated for clarity. Objects like LIMIT are also hidden from users since it does not form part of the specification. Moreover, the package body can be changed without recompiling code using it provided it meets the same specification (which is checked by the compiler).

Compile-time evaluation of expressions

In Ada the expression (1 + 1) is always equivalent to 2. Any subexpression which only involves literal values is evaluated by the compiler. In contrast, in Pascal one cannot write:

```
const
    N = 10;
    M = N + 1;
```

so one is forced to write:

```
const
    N = 10;
    M = 11; {= N + 1}
```

Such subterfuges are clearly a hinderance to program maintenance. Ada also requires the evaluation of literal expressions by the compiler when the literals are reals i.e. approximate values. This places more of a burden on the compiler (which could well use a rational arithmetic package) but is in keeping with the general philosophy of allowing the programmer to write with maximum clarity at the expense of requiring the compiler to do more work.

Generics

One consequence of the strong type mechanism is that an ordinary procedure call will not suffice since the formal and actual parameters are of a different type even though the body of the procedure required is identical. As a practical example, consider two vectors **X** and **Y** and a scaler *A*. The ability to perform **Y**:=**Y**+*A****X** is needed for a variety of scaler types: real, double length, complex, etc. Provided the scaler type has the operations '+' and '*', the body of the procedure will be correct textually even though different machine code will be needed for the various types.[9] Such a procedure is made generic having the type as a generic parameter:

```
generic
  type T is private;
  type T VECTOR is array (INTEGER range < >) of T;
  with function '+' (X, Y: T) return T is < >;
  with function '*' (X, Y: T) return T is < >;
procedure AX PLUS Y (A: in T;
                     X: in T VECTOR;
                     Y: in out T VECTOR) is
    begin . . . end;
```

The callable procedures are then constructed by instantiating the generic by inserting the appropriate types:

```
  procedure REAL AX PLUS Y is new AX PLUS Y
    (REAL, VECTOR);
```

IS ADA TOO LARGE?

The last section illustrated that the major extensions of Ada compared with Pascal are required to meet the application area of the language. Could the language nevertheless be simplified without seriously reducing the application area? Is the size of the language a serious impediment to its use?

Firstly, it would seem possible to reduce the size of the language without materially affecting the applications. However, the size reduction is marginal—no more than 10%. Moreover, the changes are bound to make programming more difficult. My own candidates for the reduction would be as follows: (1) Delete the pragma INCLUDE. This textual macro could be handled by a pre-processor, if needed. (2) Delete families of entries. This facility allows for multiple queues for services, but an additional task could do this. (3) Delete the exponentiate operator. This is not present in Pascal, but is in FORTRAN. The functionality can be provided by a generic function. (4) Delete either **mod** or **rem** since having both forms of integer division is excessive. (5) Delete the **if** statement. The case statement can be used instead and is often clearer since the conditions under which the 'else part' is executed is more explicit.

Does the size of the language make it difficult to use? For the initial user, the following aspects can be ignored: (a) Tasking: will not be used, except in ways hidden from him. (b) Generics: will only use generics already written, which is quite easy compared with producing a generic. (c) Private types: again, will only see a simple part of this as a user. (d) Real types: many applications will not need this.

On the other hand, the initial user will not be able to avoid exceptions. Exceptions have to be understood because of the clear division in Ada between the static semantics (implemented by the compiler) and the dynamic semantics (implemented by the running program). This a welcome improvement on current practice as reflected by Pascal. It is only by such understanding that the reliability of systems can be increased.

COULD PASCAL BE SIMPLER?

It might seem surprising that Pascal could be made simpler. The basic properties of Pascal that gives it its simplicity are that declarations do not require any executable code and that variables of one type occupy a fixed amount of storage. While retaining these properties, one could simplify as follows: (1) The functions eof and eoln need not have the default parameter (input) since this would then make them into ordinary Pascal functions. (2) The overloaded functions sqr and abs could be made true functions by using different names for the integer and real versions. (3) The language could be defined in terms of the ISO character set, substantially increasing portability and making the Pascal type char more adequately defined. RTL/2 takes this approach very successfully and Ada is similar. (4) The procedure 'dispose' should be removed from the language because of its inherent insecurity. The definition of the language already permits implementations to add procedures, and since compilers (for instance) need dispose, it should be

added as an option. The even less secure procedures of 'mark' and 'release' would be another alternative.

With these simplifications, perhaps one extension ought to be considered. It is particularly annoying not to be able to declare complex functions in Pascal—all functions have to return 'simple' results. With this 'extension', the language would actually be simpler for the user since an odd restriction in Pascal would be removed.

A RADICAL SUGGESTION

One reason why Ada has its current form is because of the need to make the program text as readable as possible to aid program maintenance. This assumes that program text is the sole method of communication between the programmer and the compiling system. With modern interactive computer systems, this assumption can be waived. Hence a package becomes a data-structure linked into a larger structure representing the complete library of packages. At a lower level, the user could use abbreviated keywords provided the expanded form was presented on output. Similarly, Ada is free format, but this is unnecessary for output (indeed, it is confusing). In such an environment one could design a language which has many of the attributes of Ada but is simpler as seen by the computer user.

An advantage of Pascal is that there are now a number of good textbooks on the language. The same will no doubt be true of Ada in a year or two. What would a textbook look like that was designed to teach a language based upon interactive computing? The existing books on Basic are not encouraging in terms of teaching good discipline in programming.

It can be argued that the Ada Programming Support Environment (APSE) project[10] could meet these requirements. Certainly syntax-oriented editors, pretty-printers and other tools can give the programmer a totally different view of a language. One reason for the success of Basic is the simplicity of its typical environment. Similarly, the ease of using APL, even though it needs a sophisticated character set, has contributed to its success. It is not clear how significant the APSE will be in the acceptance of Ada.

CONCLUSIONS

Although Ada is based upon Pascal, it is quite a different type of language. Pascal is excellent as an educational tool but is inappropriate for major commercial or real-time projects. Ada is more complex than Pascal but has a capability for large systems.

The choice of a language for a project is very important. Pascal and Ada together span a potentially large part of the market. In the future, tools to convert between Pascal and Ada may be available, making an initial choice less critical.[11] Currently, the use of Ada is restricted to long-term projects because of the absence of production quality compilers. Fortunately, with both US Department of Defense and EEC funding, Ada compilers are not likely to be expensive even though they will cost more than those for Pascal. A reasonable strategy at the moment would be: (a) Use Standard Pascal where possible (especially for small projects) since this allows for an upgrade into Ada, if needed. (b) Use Ada as a design language now, since packages provide a necessary framework for project management.[12]

REFERENCES

1. GPO, *Ada Language Reference Manual.* GPO %008-000-00354.8 (1980) $5.50, Superintendent of Documents, US Government Printing Office, Washington DC 20402, USA.
2. N. Wirth, The programming language Pascal. *Acta Informatica* **1**, 35–63. Springer Verlag, New York (1971).
3. US Department of Defense, *'Steelman' Department of Defense requirements for high order computer programming languages.* (1978).
4. J. G. P. Barnes, An Overview of Ada. *Software—Practice and Experience* **10**, 851–887 (1980).
5. K. Jensen and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag, New York (corrected edition 1978).
6. BSI, *Draft Standard Specification for the Computer Programming Language Pascal,* Document 79/60528 DC. (1979).
7. J. Welsh, M. J. Sneeringer and C. A. R. Hoare, Ambiguities and Insecurities in Pascal. *Software—Practice and Experience* **7**, 685–696 (1977).
8. B. A. Wichmann, *Tutorial material on the real data-types in Ada.* US Army Contract Number DAJA37-80-M-0342, National Physical Laboratory, Teddington Middlesex TW11 0LW, UK (1981).
9. S. J. Hammarling and B. A. Wichmann, (1981). *Numerical Packages in Ada.* IFIP TC2 Conference, (August 1981).
10. US Department of Defense, *Requirements for Ada Programming Support Environment.* STONEMAN (February 1980).
11. P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip and B. Krieg-Bruekner. Source-to-Source Translation: Ada to Pascal and Pascal to Ada. *SIGPLAN Notices* **15** (No. 11), 183–193 (1980).
12. B. A. Wichmann, Ada—the way ahead. *Computer Weekly,* 6–7 (6 November 1980).

Received August 1981