1983

A rational pascal

Paul A. Bailes
University of Wollongong, uow@pbailes.edu.au

A RATIONAL PASCAL

Paul A. Bailes

Department of Computing Science
University of Wollongong

Preprint No. 83-10 **November 28, 1983**

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-282-981
telex AA29022

A Rational Pascal

Paul A. Bailes

Department of Computing Science

University of Wollongong

Wollongong, N.S.W. 2500.

Australia.

ABSTRACT

Even though Pascal is a popular teaching language, it has the disadvantage that it imposes a variety of semantic and syntactic restrictions upon its users. An attempt is made to partially solve this problem by accepting the semantics of Pascal, but providing a less confining syntax for them. The replacement syntax encourages the expression of algorithms in a top-down manner. Implementation by preprocessing into Pascal is straightforward.

Keywords: Education, Pascal, Structured Programming

CR Categories: K3.2, D3.0, D2.2

October 21, 1983

A Rational Pascal

Paul A. Bailes

Department of Computing Science

University of Wollongong

Wollongong, N.S.W. 2500.

Australia.

INTRODUCTION.

Pascal (Jensen and Wirth, 1974) has been strongly promoted and adopted as a vehicle for teaching introductory programming. Yet in spite of its many advantages (Welsh, Sneeringer and Hoare, 1977) over its competitors, it exhibits certain deficiences in this regard. These stem, we feel, from the requirement that Pascal admit efficient implementation, a requirement implying that the language incorporate a number of semantic and syntactic restrictions or limitations. An example of a simple semantic restriction is that functions may not return record structures as their results, but need to return a pointer to an object, explicitly dynamically-allocated by the programmer. Examples of what we classify as syntactic limitations will follow.

We see as paramount that a language used for basic teaching should allow students to concentrate upon the task of constructing algorithms and related data structures, free of the limitations of a language. In this way do we regard Pascal as defective. As a simple improvement, we choose to accept the semantics of Pascal, and concentrate on providing for them a less confining and more suitable syntax.

PASCAL SYNTAX.

We identify the following problem areas.

(A) The use of the semicolon to separate statements confuses students and complicates the amendment of program texts. Attempts to overcome these problems by taking advantage of the "empty statement" in Pascal in order to present the semicolon as a statement terminator do not suffice. For example, while the text

```
begin
S1;
S2;
S3
end
```

may be safely re-written as

```
begin
S1;
S2;
S3;
end
```

re-writing

```
if C then
   S1
else
   S2
```

as

```
if C then
   S1;
else
   S2;
```

is syntactically incorrect. Sale (1978) develops this idea to produce a sound scheme, but at the expense of effectively introducing more syntax.

We question the need for either separators or terminators. Program legibility is best achieved by suitable formatting conventions, in this case the placement of only one statement per line. While semicolons allow an LL(1) grammar for Pascal to be given, the prevalence of more powerful parsing techniques, such as LR methods, in contemporary compiler technology (Aho and Ullman, 1977) indicates that these considerations are of diminishing importance.

(B) Similar remarks apply to a variety of symbols whose presence makes Pascal amenable to LL(1) parsing, but whose contribution to legibility under a reasonable formatting discipline, such as that of Bailes and Salvadori (1982), is minimal. For example, in the fragments

```
while C do
   S
```

and

```
if C then
   S1
else
   S2
```

keywords **do** and **then** are superfluous. On the other hand, in

```
for i := j to k do
   S
```

the ":=" and the keyword **to** play a meaningful role in visually separating the semantically-significant *i*, *j*, and *k*. Similar arguments apply in favour of the use of commas to separate elements of lists of, for example, names in variable declarations and expressions as actual parameters.

(C) The program heading serves only to redundantly nominate either the standard files *input* and *output* or files which require further declaration.

(D) The terminating period is similarly redundant.

(E) While the profusion of symbols as documented above has been to allow the simple *analysis* of Pascal programs, the *synthesis* of object code in a single pass is facilitated by placing the body of a program, procedure or function after its associated declarations. If one admits the virtues of the presentation of a program in a top-down manner, then the "main" program should appear first, with a corresponding approach for the bodies of subprograms.

(F) The syntactically-enforced ordering of the various sorts of declarations also aids one-pass compilation. However, if there exists a set of logically-related constants, types, variables and procedures and functions, as would for example embody an abstract data type, they should be permitted to appear physically related in the program text.

(G) Declarations exist to bind names to (semantic) entities. Following the principle of correspondence (Tennent, 1977), they should appear uniformly in a form such as name = entity

If the nature of the entity can be determined from its appearance alone, then the usefulness of heading keywords **const, type, var, procedure** and **function** is diminished.

(H) The precedence of operators in Pascal is inappropriate. We suggest that, as a general rule, if operator **o1** takes operands of type T1 and produces a result of type T2, and if operator **o2** takes operands of type T2 and produces a result of type T3, then **o1** should have the higher precedence of the two. This allows one to write

t_1 **o1** t_2 **o2** t_3 **o1** t_4

where the t_i are members of type T1, rather than having to write

(t_1 **o1** t_2) **o2** (t_3 **o1** t_4)

A counterexample to this policy in Pascal is where the relational operators (e.g. <) have lower precedence that the logical operators (e.g. **and**). We must write e.g.

(a < b) **and** (c < d)

NEW SYNTAX.

As suggested by the above discussion, it will be our general policy to abolish keywords and delimiters provided that:

(a) the resulting language is LR(1);

(b) simple indentation disciplines can adequately distinguish the parts of a program.

We now outline the differences between our syntax and that of Pascal. Note that the symbol S will stand for a statement in our syntax, and that S' will denote the corresponding Pascal form. Similarly will D, E, C and B denote declarations, expressions, constants and what we introduce as basic statements respectively.

A program consists of a statement *followed* by its associated declarations, the scope of which is the program. The *ordering* of declarations is *unrestricted.* We write

S

D1

.

.

.

Dn

Note that the program heading and the terminating period (C and D above) are removed.

STATEMENTS.

A statement may be one of a sequence of one or more basic statements, a selection between alternative basic statements, or a repetition of a basic statement. A basic statement is a textually atomic comprehensible form, that is

(a) an assignment

(b) a procedure call

(c) a **skip** statement

(d) an **abort** statement.

The first two are as in Pascal. The third provides a null statement, and the fourth terminates execution. The last two appear in Dijkstra (1976). No **goto** is provided, particularly as **abort** provides an effective error exit.

For selection, we write

```
if E1
   B1
elif E2
   .
   .
   .
else
   Bn
```

corresponding to the Pascal

```
if E1' then
   B1'
else
if E2' then
   .
   .
   .
else
   Bn'
```

Selection may also be expressed as a **switch** statement:

```
switch E
case C, ..., C
   B1
case C, ..., C
   B2
   .
   .
   .
case C, ..., C
   Bn
```

which corresponds to the Pascal **case** statement:

```
case E' of
   C', ..., C':
      B1';
   C', ..., C':
      B2';
   .
   .
   .
   C', ..., C':
      Bn'
   end
```

Our syntax is inspired by that of BCPL (Richards, 1969). Just as we remove keywords where they do not contribute to legibility, here we add keywords to improve it.

Iteration is expressed in one of the following forms:

```
while E
   B

repeat
   B
until E

for variable := E1 to E2
   B
```

They are not very different from the corresponding Pascal

```
while E' do
   B'

repeat
   B'
until E'

for variable := E1' to E2' do
   B'
```

We also allow the **downto** alternative of the **for** statement.

A significant property of our syntax is that only basic statements may be directly composed by the various structuring operators. If one form of composition is to be applied to the application of another, then the latter must be encapsulated in a procedure. For example, our syntax forbids

```
while E1
   if E2
         B1
   else
         B2
```

This composition can be written as

```
while E1
   select_B

select_B =
   {
   if E2
      B1
   else
      B2
   }
```

We believe that such constraints are justifiable in the introductory educational context to which this work is oriented. The rationale of top-down structured programming is the factoring of the solution of a large problem into those of sub-problems, each of which can be read and understood independently of their composition in the overall solution. The decomposition is reflected in the structured control constructs. For example, a **while** statement repeatedly executes the solution of the sub-problem which is its body, the nature of which can be separated from its iterative execution. Our syntax enforces the expression of a structured program as the composition of sub-programs by demanding that the "operands" of the control structure "operators" be expressed separately, as procedures, and be accessed by names which should be chosen to indicate their (independent) meanings.

DECLARATIONS.

As suggested in (E) and (F) above, our declarations appear generally as the form

```
name = entity
```

To aid legibility, we will allow underscores to appear in names. The keyword **var** is used effectively as a static operator applied to a type to yield a variable of that type. For example, we would write

```
person =
   record
   age = 1..limit
   name = array [1..10] of char
   end

limit = 10

first__person, second__person = var person
```

An equivalent Pascal fragment is

```
const
   limit = 10;

type
   person =
      record
      age : 1..limit;
      name : array [1..10] of char
      end;

var
   FirstPerson, SecondPerson : person;
```

Note how we remove semicolons and some keywords, and relocate another (i.e. **var**).

We declare a procedure as follows:

```
name ( formal parameters ) =
   {
   program
   }
```

The procedure body has the above form of a program - a statement followed by local declarations. For example

```
swap (x, y = var integer) =
   {
   tmp := x
   x := y
   y := tmp

   tmp = var integer
   }
```

An equivalent Pascal definition is

```
procedure swap (var x, y : integer);

   var
      tmp : integer;

   begin
   tmp := x;
   x := y;
   y := tmp
   end;
```

Note how we have a syntax for **var** parameter definitions corresponding to variable definitions. Value parameters are designated by the use of the keyword **val** rather than **var**. We believe that the nature of a parameter should be clearly distinguished.

Function definitions include the result type:

```
name ( formal parameters ) : type =
   {
   program
   }
```

As in Pascal, the function result is indicated by assignment to the function name. For example, we write

```
max (x, y = val integer) : integer =
   {
   if x > y
      max := x
   else
      max := y
   }
```

The corresponding Pascal is

```
function max (x, y : integer) : integer;
   begin
   if x > y then
      max := x
   else
      max := y
   end;
```

Finally, we allow a function body to be alternatively an expression. For example

```
square (x = val integer) : integer = x * x
```

corresponds to the Pascal

```
function square (x : integer) : integer;
   begin
   square := x * x
   end;
```

MODULES.

We have referred above to the desirability of being able to group logically-related definitions physically. On occasion, definitions will be required to be local to an abstraction, and hidden from its users. The form

```
module
   D1
   .
   .
   .
   Dn
export I1, ..., Im
```

may appear as a declaration, such that names *Ii*, declared in the declarations *Dj*, are declared in the scope of the "innermost" program or module in which the module appears. For example,

```
module
   stack = ↑stackrec

   stackrec =
      record
      stackelt = integer
      stacknxt = stack
      end

   newstack (s = var stack) =
      {
      s := nil
      }

   isempty (s = val stack) : boolean = s = nil

   top (s = val stack) : integer = s↑.stackelt

   pop (s = var stack) =
      {
      s := s↑.stacknxt
      }

   push (i = val integer, s = var stack) =
      {
      tmp := new (s)
      tmp↑.stackelt := i
      s := tmp
      }
export stack, newstack, isempty, top, pop, push
```

serves to define a *stack* (of integers) and operations *newstack*, *isempty*, *top*, *pop* and *push*. The names *stackrec*, *stackelt* and *stacknxt* are hidden.

EXPRESSIONS.

Our change is to introduce a new operator precedence satisfying (H) above. We have from lowest to highest

and, or

not

=, <>, >, >=, <, <=, **in**

+, -

*, /, **div, mod**

instead of Pascal's

=, <>, >, >=, <, <=, **in**

+, -, **or**

*, /, **div, mod, and**

not

DISCUSSION.

Several aspects of the design warrant further consideration. First, while it has been our goal to remove superfluous syntax, especially where indentation may be better used to display program structure, we use '{' ... '}' to delimit nested programs as well as indenting them. Some form of delimiting is essential in this context to avoid syntactic ambiguity, and while it is possible (Rose and Welsh, 1981) to use the start and end of indentation levels to effect this, such a mechanism gives rise to problems. One is that if our syntax were to be used as the target language of some program generator, the generator would need to generate correctly indented code. Another problem is that if we use special symbols to begin and end indentation, we need a suitably intelligent program preparation/display/edit system. It is our philosophy to give an unambiguous syntax based upon characters (possibly grouped into tokens), and treat indentation as a complementary but separate concern.

Second, having introduced the module as an information-hiding facility, why not go further to provide a data abstraction facility of the sorts proposed by Young (1981) or by Comer and Williamson (1982)?. Our answer is that we are interested in an improved syntax for an accepted set of semantics. While we regard the scopes of names as being a purely syntactic phenomenon, we regard issues dealing with types and parameterisation as being definitely semantic. Our simple information-hiding mechanism is of benefit in the framework of the existing Pascal type system.

Third, we have not scrupulously followed the principle of correspondence. Were we to, then procedure definitions would appear in a form such as

```
name = proc ( formal parameters )
   {
   program
   }
```

with a similar form for function definitions. We have chosen our suggested form because of the similarity to the mathematical notation with which introductory programming students would be likely to be familiar, as exemplified by the substitution rule of Primitive Recursive Functions (Kleene, 1952):

```
f (x1, ..., xn) =
   g (h1 (x1, ..., xn), ..., hm (x1, ..., xn))
```

Fourth, we have omitted some Pascal constructs, namely the **goto** statement, and thus labels, and procedures and functions as parameters. The first two are because of their incompatibility with the philosophy of structured programming. The case when a **goto** is clearly justified, to the end of a program when a fatal error condition is detected, is catered for by the **abort** statement. Note therefore that **abort** is not a new semantic construct, but is new syntax (a name) for one expressible in Pascal.

The latter two are omitted because Pascal deals unsatisfactorily in general with support for the advanced aspects of the functional style of programming (Backus, 1978). For example, functions may be passed as arguments to others, but not returned as results (n.b. this semantic restriction is dealt with by Georgeff (1982)). Rather than retain an half-hearted version of this facility, we choose to remove it completely. Given, once again, the introductory level at which these ideas are aimed, and current teaching practice, this deficiency cannot be considered too significant.

Fifth, we justified the requirement that all nested statements appear as separate procedures on the grounds that Structured Programming dealt with decomposing a problem into independent subproblems. However, this independence is qualified by communication via common, or global, variables. Is not our argument thus made invalid? We believe as does Backus that if one accepts the von Neumann model, then such qualifications are implicit, and that to avoid them would require investigation of an alternative model. In the meantime, our discipline provides a way of better expressing the independence that can be achieved.

Aside from the differences outlined in the preceding sections, Pascal syntax is retained (e.g. for constants and array and record accesses).

IMPLEMENTATION.

The use of Pascal to indicate the semantics of our language suggests an initial implementation strategy of translation into Pascal just as the rational FORTRAN, Ratfor (Kernighan, 1975) is implemented by preprocessing into FORTRAN. The advantage of such an approach is the simplicity of the translation. The disadvantages are that

(a) there is an added cost of translating the resulting Pascal code

(b) reliance on a simple translation means that compile-time semantic errors will not be detected until the resulting Pascal program is analysed by a translator, and error diagnostics (as with any run-time diagnostics) will be expressed in terms of this program, not in terms of the initial "Ratpas" program seen by the programmer.

Nevertheless there are merits in producing a quick implementation (e.g. for experiments with the new language).

The first issue of the translation is the insertion or replacement of delimiters, which is clearly trivial. Of more significance is the re-ordering of declarations. Our

statement declaration ... declaration

has to be expressed in Pascal as

declaration ... declaration compound-statement

and, for the list of declarations, the correct ordering of the sorts of declaration (constants, types, variables and procedures and functions) must be achieved. Furthermore, implementations of Pascal often demand that a name be declared prior to its use. Cyclic definitions may occur only within the following categories:

(a) types;

(b) procedures or functions.

For each, a dependency graph is constructed. If cycles occur, then we either report an error or take advantage of the facilities Pascal provides for cyclic definitions. For procedures and functions, **forward** definitions are standard.

For example, the fragment

```
P1 (...) =
    {
    .
    .
    .
    P2 (...)
    .
    .
    .
    }
P2 (...) =
    {
    .
    .
    .
    P1 (...)
    .
    .
    .
    }
```

is translated

```
procedure P2 (...);
   forward;

procedure P1 (...);
   begin
   .
   .
   .
   P2 (...)
   .
   .
   .
   end;

procedure P2 ;
   begin
   .
   .
   .
   P1 (...)
   .
   .
   .
   end;
```

The fragment

```
element =
   record
   dat = integer
   nxt = eptr
   end

eptr = ↑element
```

becomes

```
type
   eptr = ↑element;

   element =
   record
   dat : integer;
   nxt : eptr
   end;
```

where forward references to pointers to types are allowed. However

```
T1 = array [1..10] of T2

T2 = T1
```

is detected as erroneous.

Because we allow underscores to appear in names, and because Pascal implementations frequently limit the length of identifiers, the preprocessor must rename them and produce output according to some standard e.g. *N1* for the first encountered, *N2* for the second, etc.

The **skip** statement is implemented by a call to a predefined no-op procedure; **abort** is effected by a jump to an inserted terminating label.

A detailed account of an implementation is given by Shepanski (1983).

DETAILED EXAMPLE.

The problem is to read a list of not more than 1000 numbers and output them in ascending order.

The solution in our syntax is as follows.

```
initialise
input__the__numbers
sort__them__and__output

input__the__numbers =
   {
   while numbers__left
      read__into__list

   numbers__left : boolean = not eof

   read__into__list =
      {
      numbers__read := numbers__read + 1
      readln (table [numbers__read])
      }
   }

sort__them__and__output =
   {
   for i := 1 to numbers__read
      select__smallest__from (i)

   i = var minrange

   select__smallest__from (base = val minrange) =
      {
      m := index__of__smallest__from (base)
      writeln (table [m])
      table [m] := table [base]

      m = var minrange

      index__of__smallest__from (base = val minrange) : integer =
         {
         if base = numbers__read
            index__of__smallest__from := base
         else
            index__of__smallest__from := test__base (index__of__smallest__from (base + 1))

         test__base (index = val minrange) : minrange =
            {
            if table [base] < table [index]
               test__base := base
            else
               test__base := index
            }
         }
      }
   }
```

```
initialise =
   {
   numbers_read := 0
   }

minrange = 1..limit
limit = 1000

table = var array [minrange] of integer
numbers_read = var 0..limit
```

A structurally-equivalent Pascal program is

```
program example (input, output);

const
   limit = 1000;

type
   minrange = 1..limit;

var
   table : array [minrange] of integer;
   NumbersRead : 0..limit;

procedure InputTheNumbers;

   function NumbersLeft : boolean;
      begin
      NumbersLeft := not eof;
      end;

   procedure ReadIntoList;
      begin
      NumbersRead := NumbersRead + 1;
      readln (table [NumbersRead])
      end;

   begin
   while NumbersLeft do
      ReadIntoList
   end;

procedure SortThemAndOutput;

   var
      i : minrange;

   procedure SelectSmallestFrom (base : minrange);

      var
         m : minrange;

      function IndexOfSmallestFrom (base : minrange) : integer;

         function TestBase (index : minrange) : minrange;
            begin
            if table [base] < table [index] then
               TestBase := base
            else
              TestBase := index
            end;

         begin
         if base = NumbersRead then
            IndexOfSmallestFrom := base
         else
            IndexOfSmallestFrom := TestBase (IndexOfSmallestFrom (base + 1))
         end;

      begin
      m := IndexOfSmallestFrom (base);
```

```
            writeln (table [m]);
            table [m] := table [base]
            end;

        begin
        for i := 1 to NumbersRead do
            SelectSmallestFrom (i)
        end;

    procedure initialise;
        begin
        NumbersRead := 0
        end;

    begin
    initialise;
    InputTheNumbers;
    SortThemAndOutput
    end.
```

As Hanson (1981) points out, "real" programs are not usually displayed in a variety of fonts. Therefore we have presented these programs in a corresponding manner for comparative purposes.

CONCLUSIONS.

We have offered what we believe to be a simple remedy to some of the purely syntactic drawbacks of Pascal. No doubt these suggestions will provoke disagreement. One obvious area of improvement is to incorporate the semantics of the ISO Pascal Standard (Standards Association of Australia, 1983). The definite lesson that can be learned, however, is that alternatives to Pascal exist and are worth consideration.

ACKNOWLEDGEMENTS.

The helpful comments made by the anonymous referees on the earlier version of this paper are sincerely and gratefully acknowledged, as well as the comments of, and interest shown by, Cecily Bailes, Jan Hext, John Lions, Philip Maker, Ross Nealon, Juris Reinfelds, Arthur Sale and Michael Shepanski.

REFERENCES.

Aho, A.V. and Ullman, J.D. (1977), "Principles of Compiler Design", Addison-Wesley.

Backus, J. (1978), "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", CACM vol. 21, no. 8, 613-641.

Bailes, P.A. and Salvadori, A. (1982), "A Semantically-based Formatting Discipline for Pascal", University of Wollongong Department of Computing Science Preprint 82/19 (to appear in Software - Practice and Experience).

Comer, D. and Williamson, K. (1982), "An Alternative to Young's Module Facility for Pascal", Software - Practice and Experience, vol. 12, no. 10, 907-913.

Dijkstra, E.W. (1976), "A Discipline of Programming", Prentice-Hall.

Georgeff, M.P. (1982), "Evaluating Typed Lambda Expressions using a Stack Machine", Austrln. Comp. Sci. Commun., vol. 4, no. 1, 266-276.

Hanson, D.R. (1981), "Is Block Structure Necessary?", Software - Practice and Experience, vol. 11, no. 8, 853-866.

Jensen, K. and Wirth, N. (1974), "Pascal User Manual and Report", Lecture Notes in Computer Science, vol. 18, Springer.

Kernighan, B.W. (1975), "RATFOR - A preprocessor for a Rational Fortran", Software - Practice and Experience, vol 5, no.4, 395-406

Kleene, S.C. (1952), "Introduction to Metamathematics", Van Nostrand Reinhold.

Richards, M. (1969), "BCPL - A tool for compiler writing and systems programming", Proc. Spring Joint Comp. Conf. 1969, 557-566.

Rose, G.A. and Welsh, J. (1981), "Formatted Programming Languages", Software - Practice and Experience, vol. 11, no. 7, 651-670.

Sale, A. (1978), "Stylistics in Languages with Compound Statements", Aust. Comp. J., vol. 10, no.2, 58-59.

Shepanski, M.P. (1983), "Ratpas Documents", University of Wollongong Department of Computing Science.

Standards Association of Australia (1983), "Programming Language Pascal", Australian

Standard 2580-1983.

Tennent, R.D. (1977), "Language Design Methods Based on Semantic Principles", Acta Informatica, vol. 8, 97-112.

Welsh, J., Sneeringer, W.J. and Hoare, C.A.R. (1977), "Ambiguities and Insecurities in Pascal", Software - Practice and Experience, vol. 7,

Young, M. (1981), "Improving the Structure of Large Pascal Programs", Software - Practice and Experience, vol. 11, no. 9, 913-927.