1982

A rational pascal

Paul A. Bailes
University of Wollongong

A RATIONAL PASCAL

by

Paul A. Bailes

Preprint No. 82-20 November 12, 1982

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
PO Box 1144
Wollongong NSW 2500

ABSTRACT

Discussed is the way in which even though Pascal is used for teaching programming, it is unsuitable as a tool for the development of algorithms (as a human thought process) because of the burden of syntactic detail imposed upon the programmer. Consequently, educators sometimes introduce a pseudo-code in which to derive and express algorithms. Such a notation is free of any syntactic detail, but is unfortunately free of any rigorous semantics definition, with consequent problems in using it to devise and define algorithms.

An attempt is made to provide a technical solution to the problem by providing a less obtrusive syntax for Pascal semantics, providing a vehicle for expression based on a set of well-defined constructs but free of the superfluous notational detail that afflicts Pascal programs.

The results of the attempt are assessed, and consideration given to implementation issues.

Keywords: Structured Programming, Pascal, Education.

CR Categories: 1.5, 4.12, 4.20.

A Rational Pascal

Paul A. Bailes

Department of Computing Science
University of Wollongong
PO Box 1144
Wollongong NSW 2500

THE PROBLEM

The Pascal language (Jensen & Wirth, 1974) is strongly promoted as a vehicle for introductory programming teaching, yet in spite of its advantages (Welsh ,Sneeringer and Hoare, 1977) over the "competition" (including FORTRAN, PL/I, BASIC, Algol-60, COBOL and Algol-68) it exhibits severe deficiencies in this regard.

This is brought out by an analysis of the programming process, particularly in an educational situation where the detail and separateness of a variety of concerns should be made obvious. The particular matter that concerns us is the distinction between what can be called the **design** of an algorithm and its **implementation** in a particular programming language.

By design, we mean the way in which (in this setting) an algorithm is specified by a sequence of refinements of abstractions (according to the "top-down" design methodology (Wirth, 1971)) and the composition of the abstractions by way of the mechanisms of Structured Programming (Dahl, Dijkstra and Hoare, 1972) namely repetition, selection and sequencing but without reference to particular features (read "restrictions"), both syntactic and semantic, imposed by particular languages. An example of a syntactic restriction in Pascal is the compulsory placement of a semicolon character (';') between each statement; an example of a semantic restriction being the fact that functions may only return simple (non-structured) results. Implementation refers to the coding of an appropriate design into a chosen language, in the process of which such relatively low-level details may be given attention.

The advantage of this separation of concerns is that the significant intellectual effort (algorithm design) has been simplified by the removal of this detail. The disadvantage, however, is that the language in which this significant discourse takes place is not well-defined. For example, a practice familiar to us involves the introduction of a "pseudo-code" for algorithm development combining the above-mentioned structured constructs with the implied characteristics of a von Neumann machine and miscellaneous conditions and operations expressed in English. When these conditions and operations are not seen to have a clear implementation in a programming language (e.g. Pascal), they may be subject to a further process of explication. For example, let us consider the design of an algorithm to compute the gcd of each of a list of pairs of numbers:

```
gcd of pairs =
  while pairs left do
    read a pair
    compute gcd of pair
    print it
```

In this particular format, the first line provides a title for the algorithm (by which name it may be referred to at some "higher level" of a larger design process); the second signifies iteration of the next three lines (grouped by common indentation) as long as an input data element (a number-pair) is available; while these three (iterated) lines perform the indicated functions.

Now "read a pair" and "print it" seem at first glance to require no further expansion, so we shall deal with the historically non-trivial "compute gcd of pair":

```
compute gcd of pair =
  while first element of pair ◇ second element of pair do
    if second element > first element then
      swap them
    set first element to first element minus second element
  result is first element
```

Educationally, expansion of "swap them" would depend upon the state of advancement of the class: at an early stage of development, it would be an ideal candidate; but perhaps by even the "gcd" level, it may be assumed to be sufficiently primitive. While

there are a number of "loose ends" such as the association between the "it" in "print it" and the "result" determined in the expansion of "gcd", it is felt that a reasonable understanding of the English language allows for the unambiguity necessary to call the above an algorithm, the following Pascal **procedure** being trivially derived:

```
procedure gcd_of_pairs;
 var pair1,          (* first element *)
    pair2,          (* second element *)
    tmp: integer;    (* temporary for swap; all integers *)
 begin
  while not eof do
  begin
   read(pair1,pair2);        (* read a pair *)
   while pair1 <> pair2 do   (* composite gcd of pair *)
   begin
    if pair2 > pair1 then
    begin
     tmp := pair1;          (* swap them *)
     pair1 := pair2;
     pair2 := tmp;
    end;
    pair1 := pair1 - pair2  (* set etc. ... *)
   end
   writeln(pair1)
  end
 end;
```

Note how (at least potential) abstractions "compute gcd of pair" and "swap them" have been expanded in-line, but could have been made into procedures to make the top-down development of the ***algorithm*** more apparent in the ***program***. Also to be noted are

(i) the assumption of the roles of "result" and "it" by "pair1"

(ii) the general profusion of (mostly syntactical) detail: semicolons, **begin** ... **end** pairs, particular symbols for certain operations.

In spite of this encouraging experience, there is still cause for concern regarding

(a) how introductory students react to performing algorithm design in an only intuitively defined language, and

(b) the trend toward formality in algorithm design propounded by such as Dijkstra (1976) and Gries (1981), a formality only enjoyable in the context of a well-defined language of discourse.

To understand the first requires one to put oneself in the position of a "new" student, totally unfamiliar with computers, programming languages and the design of algorithms. There is no particular reason to suggest that natural language (the basis of the pseudo-code) should be of any benefit in the design of algorithms. The general argument that as people are familiar with using natural language, they should then design algorithms using such a familiar language would seem to presuppose a belief that as natural language, reflecting human thought processes, is suitable for writing algorithms, then natural languages were the product of algorithmic thought processes. Our admittedly limited acquaintance with anthropology not withstanding, it does appear that the development of highly organised societies requiring the execution of certain processes (as defined by **well-designed** algorithms) post-dates the "invention" of natural language, so that the influence of algorithmic concepts on such development is negligible. Consequently, natural language cannot be accepted, *ipso facto*, as the most suitable algorithmic language.

Granted, the described pseudo-language incorporates some well-defined concepts (those for Structured Programming) but experience shows they tend to become "submerged" by the use of natural language to describe operations and conditions, which to the uninformed are neither unambiguously defined nor in any sense "primitive". Furthermore, having developed such an "algorithm", the (student) programmer is now faced with a tedious error-prone "coding" exercise (into the language of choice).

A SOLUTION

Fortunately, the doctrine of top-down development provides a way of unifying the formality of a programming language with the creativity-inspiring freedom of natural language, by simply allowing the English-language description of a condition or

operation to act as the name of a **procedure** or **function** awaiting further definition. The process of algorithm **design** ultimately must rest on the properties of the chosen programming language, supplying the necessary direction of purpose lurking behind the informal approach, and supplying the formality needed for the various verification/synthesis schemes referred to. Furthermore, at any stage in the top-down development, expansion of a "sub-program" (so-called) may be defined, but still leaving a valid program fragment. Note that we are explicitly accepting the basic operations and data types of a chosen language.

Unfortunately, having established the virtue of semantic detail, for the purposes of providing much-needed ultimate formality, we see that the syntax used for an exercise of the above kind brings with itself its own problems - those of totally irrelevant detail. As pointed out by Dijkstra (1976) and Hoare (1981), the language used to solve the problem becomes part of the problem.

We therefore embark upon a language design experiment aimed at capturing a well-defined semantics in an environment of minimal syntactic interference. Because we wish to clearly limit the scope of this exercise, and so avoid a necessarily lengthy development of, say, theories of language design and programming education, we choose to accept without question the semantics of a particular existing language, to wit, Pascal. This choice is made because

(a) its semantics are both well-known and closely correspond to the characteristics of the von Neumann machine implied by the pseudo-codes which inspired this exercise (i.e. data types and control constructs)

(b) it is itself a popular teaching language, and our experiment can be seen as a contribution to the development of Pascal culture.

What are the problems, then, with the syntax used to express these desirable semantics? We initially re-state the general criticism levelled by Habermann (1973) that for a teaching language, requiring much "kindness" to its users, the design goals of Pascal

including efficiency at both compilation and execution time of Pascal programs would seem to be contradictory to this need. An example of a semantic restriction inspired by the need for execution-time efficiency is the common restriction on set type sizes, but as we have chosen to accept a language's (in this case, Pascal's) semantics, such issues will be here ignored. Examples of syntactic restriction brought about by the need for compile-time efficiency are now discussed.

We identify the following items:

(i) The placement of semicolons is the cause for considerable confusion amongst students not familiar with the properties of context-free grammars and the formal syntactic definition of Pascal. Attempts (by instructors) to take advantage of the presence of the "empty statement" in the Pascal syntax to present the semicolon as a simple statement **terminator** (rather than the more conceptually complex statement **separator** it really is), say:

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

becomes

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

in which the empty statement is separated from S3 by ';' and is followed by **end**, are doomed to failure in the context of

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

becoming

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

which is syntactically incorrect.

Simple removal of the semicolon from the language would altogether remove

this irritation. We dispose of the alleged advantages of the semicolon, that it aids comprehension and compilation as follows. The case for the semicolon aiding legibility only holds true if more than one statement appears on a line, which rarely occurs, which we propose is bad style in any case, and even if it does occur, legibility is more aided by judicious spacing than by an intrusive semicolon.

With regard to compilation, removal of the semicolon removes the LL (or one-character look-ahead) property found so desirable by Wirth, but the resultant language is still LR(1) (Knuth, 1965). In view of the prevalence of the LR parser in contemporary compiler technology (Aho and Ullman, 1977), we see that abolishing the semicolon and other delimiters generally is no great loss.

(ii) The program heading contributes no meaningful information to the program, represents a further opportunity for syntactic error, and should be removed.

(iii) So should the terminating period.

(iv) Declarations contain "noise words" such as **const**, while the relevant information can often be determined from the context e.g. given as a declaration

```
size = 5
```

it is clear that a constant definition is indicated; to say

```
const size = 5
```

contributes nothing. Admittedly, the **const** keyword plays a role in the establishment of a declarative context (as do **type**, **var**, **procedure**, **function** and **label**), so we shall endeavour to find a simpler way of establishing this context.

(v) In company with (iv), declarations perform a single purpose - the binding of names to entities, be they values (for constants) or functions. In the **spirit** of the

principles of correspondence and abstraction (Tennent, 1977) (shown to be fully observable by Pascal only by dint of significant **semantic** extension), a single name-entity binding syntax should be introduced.

(vi) The process of top-down development involves the identification of abstractions by their uses prior to their expansions or definitions. The exigencies of one-pass compilation (for "efficiency") adopted by Pascal forbid the expression of an algorithm in this natural way. Particular details are the necessity (on occasion) for **forward** declarations for procedures and functions, and the careful ordering of type declarations.

It is now our task to synthesise a language design to avoid these problems.

DESIGN DETAILS.

We present a more rational syntax for Pascal semantics than that provided by Pascal itself. The metalanguage is a simple variant of "standard" context-free notation (BNF (Backus, 1959) being an oft-occurring variant) as follows:

(i) the symbol -> is used for production

(ii) the symbol | is used for alternation

(iii) the empty symbol (or simple juxtaposition) denotes concatenation

(iv) [X] means X is optional

(v) X^+ means X occurs one or more times

(vi) X^* means $[X^+]$ (Kleene Star)

The precedence of operations is (highest to lowest): concatenation; alternation; repetition (* or +). Parentheses (...) may be used for grouping e.g.

XYZ^+ includes XYZZZ...
$(XYZ)^+$ includes XYZXYZXYZ...

THE LANGUAGE

program -> statement declaration*

A program is a statement followed by a (possibly empty) list of declarations, the scope of the declared names being the entire program.

```
statement ->
        selection_statement
        | repetition_statement
        | basic_statement+
```

A statement is either a selection_statement (structured selection), a repetition_statement (structured repetition) or a list of basic_statements (structured sequencing). The non-emptiness of the latter involves no semantic problems because an explicit no-op or skip statement is provided (see below).

```
selection_statement ->
        conditional_statement
        | case_statement

conditional_statement ->
        if guarded_statement
        (elif guarded_statement)*
        [else
           basic_statement]

guarded_statement ->
        expression
           basic_statement
```

A conditional_statement is, essentially, an ordered list of guarded_statements, such that the first expression (as defined by the above syntax) in that list evaluating to true, rather than false, implies execution of the corresponding basic_statement. If none such does, and the **else** part is present, then that basic_statement is executed. The guarded_statement is introduced as a purely syntactic phenomenon, no relationship with some more sophisticated semantics (Dijkstra, 1976) being implied. Given that E denotes an expression and B a basic_statement, a transformation T from this language to Pascal may be defined, in this case, given

```
if E
     B
elif E
     B
  .
  .
  .
else
     B
```

T produces (Pascal)

```
if T(E) then T(B)
else
if T(E) then T(B)
     .
     .
     .
else T(B)
```

where the effects of T when applied to expression (E) and basic_statement (B) are yet to be defined.

```
case_statement  ->
          case expression
            case_branch+
          [else basic_statement]

case_branch -> constant+ : basic_statement
```

A case_statement is an ordered list of case_branches, perhaps followed by an **else** part. The expression is evaluated, and for the first constant in the list of case_branches equal to it, its corresponding basic_statement is executed. If none does, and the **else** part is present, then its basic_statement is executed, otherwise nothing happens (equivalent to **skip**, see below). In terms of the translation T, and given that C is a constant, the fragment

```
case E
  C ... C : B
     .
     .
     .
  C ... C : B
  else B
```

transforms into

```
case T(E) of
  T(C), ... ,T(C) : T(B);
        .
        .
        .
  T(C), ... ,T(C) : T(B);
  others: T(B)
end
```

where the application of T to a constant is yet to be defined, and assuming for the purposes of exposition the availability, as in some implementations of Pascal (e.g. that for the DEC-10), of an others or default case branch.

```
repetition_statement  ->
            while_statement

            | repeat_statement

            | for_statement

while_statement  ->
            while expression
              basic_statement

repeat_statement  ->
            repeat
              basic_statement
            until expression

for_statement  ->
            for variable := expression (to | downto) expression
              basic_statement
```

The following transformations apply:

(a)

```
        while E
            B
```

becomes

```
        while T(E) do
            T(B)
```

(b)

```
        repeat
            B
        until E
```

becomes

```
repeat
    T(B)
until T(E)
```

(reflecting the syntactic economy of the Pascal repeat construct)

(c)

```
for V := T(E) to T(E)
    T(B)
```

becomes

```
for V := E to E do
    B
```

```
basic_statement ->
        assignment_statement

        | call

        | return expression

        | skip

        | abort
```

A basic_statement is one of a selection of primitive constructs: *assignment_statement*; (procedure) call (both explained later); **return** expression, denoting the result of a function invocation (see below); **skip** the null statement; and **abort**, execution of which terminates program execution.

assignment_statement -> *variable* := *expression*

The value of an expression may be assigned to a variable just as in Pascal.

call -> identifier [(actual_parameter^{+})]

A (procedure) call (also, a function call) comprises an identifier (to identify the procedure/function being invoked), and (optionally) a list of actual_parameters enclosed by parentheses.

actual_parameter -> variable | expression

An actual_parameter can be either a variable or expression depending on whether or not it corresponds to a variable or value formal_parameter. For an example

```
sort ( a b )
```

applies procedure (or function) sort to arguments a, b. Note the omission of a comma separating a, b. Because of the intuitive nature of these concepts, as well as **skip** and **abort**, no transformation into Pascal for definitional purposes is given (but see **IMPLEMENTATION** below). The mechanism for returning a value from a function invocation is discussed with respect to declaration, which now follows.

```
declaration  ->
            simple_declaration

            | simple_function

            | procedure

            | function

simple_declaration  ->  name+  =  entity

entity  ->
            constant
            | type
            | var type
```

The entity to which a name (or names) is bound by a simple-declaration may be either a constant, a type, or a variable of an indicated type. Given that T denotes a type (C a *constant as above), the fragment*

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

becomes (in Pascal)

```
const limit = 10;
type person =
   record
      age : 1..limit;
      name : array [1..10] of char
   end;
var
   jim, fred : person;
```

Note that the binding of a name to a record element e.g.

```
age = 1..limit
```

does not require the use of the **var** keyword because that name can only be bound to a variable (element of a particular record).

```
simple_function  ->
        name [ ( formal_parameter_specification+ ) ] : type =
          expression

formal_parameter_specification -> name+ = parameter_kind  type

parameter_kind -> var | val

procedure  ->
        name [ ( formal_parameter_specification+ ) ] = {
            program
        }

function  ->
        name [ ( formal_parameter_specification+ ) ] : type = {
            program
        }
```

A simple_function represents the binding of the given name to a function whose body is the expression in which the names declared as parameters in the formal_parameter_specification are bound. A procedure or function likewise defines a procedure or a function whose body is the program, the essential difference between the body of a function or a procedure being determined by whether or not the syntactically last statement(s) executed by the (sub-) program is (are) a **return** statement(s). A formal_parameter_specification declares as a formal parameter the variable name(s) of the given type, using keywords **var** and **val** to distinguish between variable and value parameters. As usual, we present a code fragment and its translation:

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

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

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

    tmp = var integer
}
```

becomes in Pascal

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

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

procedure swap(var x, y : integer);
var tmp:integer;
begin
    tmp := x;
    x:= y;
    y:= tmp
end;
```

Note that the call of a procedure in our new notation differs to no great extent from that of Pascal.

Terms referred to but not defined (e.g. constant, expression) may be assumed to be identical to the corresponding Pascal phenomena (e.g. T(E) => E), noting the detail that the syntax of a function call corresponds to that for a (procedure) call as specified above.

DISCUSSION.

We reiterate the motivating principle - to produce a language embodying the semantic features of Pascal but with a less obtrusive syntax, perhaps at the expense of efficient compilation. It is with respect to this that the various technical features of, and decisions embodied in, the design are to be assessed.

The various unnecessary delimiter characters and related phenomena criticized earlier have been removed, including the commas separating names in variable declarations and parameter lists. The LISP (McCarthy, 1960) experience shows that this particular detail can be dispensed with comfortably.

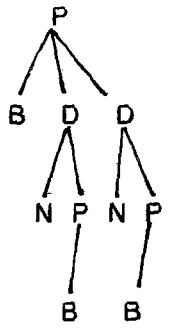
Unfortunately, there is an area of the language where some notion of delimiting is required. When a context-free grammar embodies (directly or indirectly) rules of the form

$$P \rightarrow B\ D^*$$
$$D \rightarrow N\ P$$

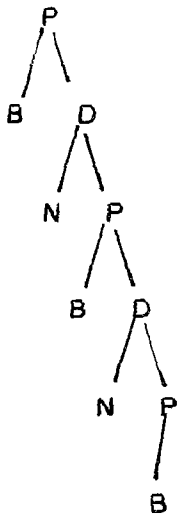
an ambiguity is introduced, exemplified by the sentence

BNBNB

which admits the parse trees



and



In the case of our syntax, P corresponds to program, B to basic_statement (a terminal symbol for the purposes of this discussion) D to a declaration (of a procedure or function) and N to the name declared by a declaration. The ambiguity may be removed by the introduction of delimiters around the recursion, say

P -> B D*
D -> N (P)

so that the senses conveyed by the above parse trees are given by the respective sentences

B N(B) N(B)

and

B N(B N(B))

Now, our desire for unobtrusive syntax has been motivated by the possibility of formalising the pseudo-code used to develop algorithms, in which the delimiting process is carried out by relative indentation, and which is adapted by "pretty printing" programs for various "structured" languages (Pascal and C included). There has been proposed a scheme cast in the mould of a variation to Pascal in which the **begin end** delimiter pairs of that language are replaced by indentation (Rose and Welsh, 1981) but which for practical use requires some form of "smart" editor or program-entry system. Because we wish our product to be available in a "low-tech" manner, we have

avoided this solution and opted for an explicit delimiter pair: (...). The unobtrusiveness of this form, together with a stylistic discipline of associating with it indentation should help to overcome the failure to fully meet our goal.

This slight disadvantage has been mitigated even further by the removal of a related recursive component of Pascal syntax, the compound or nested statement (or block in Algol-60 terminology). In our language, a program can generate only, say, a sequence of basic_statements, none of which can directly be expressed in terms of some (other) form of composition (sequencing again, or selection or repetition) but must be expressed via a (procedure) call and its declaration to the associated program. Methodological issues came into play here, it being our policy to enforce such disciplines as present themselves by way of purely syntactic manipulation. The value of structured programming is in the way in which separate program components can be understood in isolation. This logical isolation is, we suggest, aided by textual isolation. Furthermore, understanding of a phenomenon is retained by the association with it of some outstanding "key", for which we propose a meaningful, descriptive identifier. Thus, in reading a program in our syntax top-down, one comes across a meaningful identifier in its use, but whose name (and possibly context) would provide clues to its definition. In a bottom-up scan, the definition is reinforced by the meaningful name. (Note that, unlike Pascal, the words top-down and bottom-up here refer to both the physical and logical aspects of program understanding). It is important to realise that we are not simply just re-hashing the usual arguments for using well-chosen names in programs, but are proposing a discipline supporting the structured programming doctrine of understanding programs one portion at a time. Given that a basic_statement represents some intuitively-understood concept (be it because it is an inherently primitive operation such as an assignment statement, or because it is a call of a procedure with the process of intuition as suggested above), it is susceptible to only one form of Structured composition (repetition, selection,sequencing) before

being given an intuitive handle (a name) as a prelude to the understanding of compositions in which it is used.

The syntax for declarations has been chosen to reflect the simple name-entity binding that so takes place, in the manner we believe pseudo-codes tend to adopt. The simplicity of the syntax tends to obey the principles of Abstraction and Correspondence. Pragmatically, it is felt that the class of entity bound to a name is clear from the "expression" on the "right-hand-side" of the declaration.

The keyword **val** was added to formal_parameter_specifications because of the apparent confusion caused to students by the default pertaining to the omission of **var**. In a "strongly-typed" environment, compulsion to specify clearly the nature of a parameter should find sympathy.

The simp_function_declaration is included to overcome the Pascal

```
function X( args ) : type
begin
  X := expression
end
```

idiom to implement a parameterised expression to give

```
X( args ) : type = expression
```

It is necessary to even in this simple case distinguish between **var** and **val** parameters because the expression (body) may involve a call to a function which is capable of altering one (or more) of its possibly **var** parameters. It will be observed that the parameters used in a procedure or a function are declared at the head of the procedure or function and **not** after their use in it. This is so because, like the type of a function, they are part of its interface with the environment of its use, and so must be clearly specified **prior** to the implementation of the procedure or function, which our syntax reinforces.

RESTRICTIONS.

On the large scale of language design and programming methodology issues, notable is our omission of any reference to data abstraction or information hiding mechanisms, given that the unspoken norm of language rationalisation exercises such as that which we have effectively carried out is to provide a new "feature" hitherto unavailable in the rationalised language and that the named facilities are noticeably missing from Pascal. Our justification is our self-imposed restriction to maintaining the semantic basis of the Pascal language, and that an acceptable (to our standards) treatment of data abstraction could not be carried out in that restricted framework. In fact, the work of Tennent (1977) commented on by Berry (1981) indicates the necessity for an environment as rich as Algol-68, which would seem technically unacceptable. Our own particular hypothesis is that the complexity of Algol-68 in its attempt at generality and uniformity is merely a reflection of the complexity of the von Neumann model (as argued by Backus (1978)). At any rate, the topic of data abstraction has been deemed outside the scope of this discussion.

The omission of functions and procedures as parameters is justified on two grounds. Firstly, the sort of general abstraction capability hinted at by this facility far exceeds the normal run of Pascal semantics, so we aim for a consistent set of abstraction tools. In view of Backus' criticisms (Backus, 1978) of the general abstraction facilities of the lambda-calculus (Church, 1941), our restriction to simple abstraction of data, and not operations seems at least justifiable. Secondly, at an introductory level, the idea of a procedure or a function as a function in its own right which can be abstracted out of another is perhaps too subtle. Certainly, the syntax of Pascal, which we have rationalised, discourages any such line of thought.

Finally, the omission of the **goto** statement is because its use directly contradicts the top-down, structured programming methodology supported by this language.

IMPLEMENTATION.

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

(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 without some effort in this regard, 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.

That is to say, the best implementation, particularly for teaching purposes is a dedicated one.

Nevertheless, if problem (b) can be overcome, there are merits in producing a quick implementation (e.g for experiments with the new language); a general solution to the problem is indicated by the way in which the C language (Kernighan and Ritchie, 1978) allows line-number assertions so that C programs produced by pre-processors (e.g. Yacc (Johnson, 1978)) may generate line-numbered diagnostics with reference to the original program.

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

program declaration *

has to be expressed in Pascal as

declaration * block

and for the list of declarations, the correct ordering of the sorts of declaration (constants, types, variables, procedures and functions (no labels!)) and of the declarations

of each sort must be achieved. We observe that in Pascal, declarations of each sort can only depend on a "preceding" sort (e.g. a variable declaration will refer to a type, whereas a type declaration cannot refer to a variable) except for type and procedure or function declarations, which can access entities of the same sort declared in the same set of declarations. We construct a dependency graph for

(a) types, and

(b) procedures or functions

If a cycle is found to exist, then in case (b) a forward declaration is made (assuming our Pascal implementation supports them); in case (a) we determine whether or not it involves the single allowed form of forward reference (to a pointer of the type's base type); if not, then we must report an error.

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;
```

But

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

T2 = T1
```

is detected as erroneous.

Our liberal use of long identifiers using an underscore ('_') for legibility results in the necessity for renaming them to some standard (e.g N00001 etc...).

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

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 = not eof

   read_into_list = {
      numbers_read := numbers_read+1
      read (table[numbers_read])
   }
}

sort_them_and_output = {
   for i := 1 to numbers_read
      select_smallest_from (i)

   i = var numrange

   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
            return base
         else
            return test_base (index_of_smallest_from (base+1))

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

initialise = {
   numbers_read := 0
}

minrange = 1..limit
limit = 1000

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

A corresponding Pascal program (ignoring limits on identifier sizes) is

```
program example (input, output);

const limit = 1000;

type minrange = 1..limit;

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

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

procedure select_smallest_from (base : minrange);
   var m : minrange;

   function index_of_smallest_from (base : minrange) : integer;
      var index : minrange;

      begin
         if base = numbers_read then
            index_of_smallest_from := base
         else
         begin
            index := index_of_smallest_from (base+1);
            if table[base] < table[index] then
               index_of_smallest_from := base
            else
               index_of_smallest_from := index
         end
      end;

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

begin
   numbers_read := 0;
   while numbers_left do
   begin
      numbers_read := numbers_read+1;
      read (table[numbers_read])
   end;

   for i := 1 to numbers_read do
      select_smallest_from (i)

end.
```

Note that because, as Hanson (1981) points out, "real" programs are not usually

displayed in a variety of fonts, we have presented these programs in a corresponding manner for comparative purposes.

CONCLUSIONS

The advantages of our rationalisation are as follows :

(a) we have succeeded in removing the unnecessary syntactic detail that inhibits the top-down development of an algorithm in Pascal (an important aspect of which was the need to write down declarations and specifications prior to the discovery of the need for them, in contradiction of the top-down approach)

(b) the tree-structured record of top-down development is retained in the program text, with procedure and function names clearly identifying the nodes of the tree, this discipline being enforced.

The possible disadvantage, the removal of helpful delimiters and keywords, is countered by the awareness that the prominence of keywords in published algorithms and programs is not enjoyed in a practical environment where bold typefaces are not available, and that indentation, which we advocate, is a better visual aid to program structure.

We conclude that it is possible to express the semantics of Pascal *sans* excessive syntactic detail, and that by virtue of the improvements made, the process of algorithm development **can** be carried out in the environment of a well-defined language, without reference to necessarily ill-defined pseudo-codes.

REFERENCES

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

Backus, J. (1959), "The syntax and semantics of the proposed international algebraic language of the Zurich ACM - GAMM conference", Proceedings of the International Conference on Information Processing, UNESCO, 125-132.

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.

Berry, D.M. (1981), "Remarks on R.D. Tennent's Language Design Methods Based on Semantic Principles: Algol68, A Language Designed Using Semantic Principles", Acta

Informatica, vol. 15, 83-98.

Church, A. (1941), "The Calculi of lambda-conversion", Princeton University Press.

Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. (1972), "Structured Programming", Academic Press.

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

Gries, D. (1981), "The Science of Programming", Springer.

Habermann, A.N. (1973), "Critical Comments on the programming language Pascal", Acta Informatica, vol. 3, 47-57.

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

Hoare, C.A.R. (1981), "The Emperor's Old Clothes", CACM, vol. 24, no. 2, 75-83.

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

Johnson, S.C. (1978), "Yacc: Yet Another Compiler-Compiler", UNIX* Programmer's Manual, Seventh Edition, Volume 2B.

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

Kernighan, B.W. and Ritchie, D.M. (1978), "The C Programming Language", Prentice-Hall.

Knuth, D.E. (1965), "On the Translation of Languages from Left to Right", Information and Control, vol. 8, no. 6, 607-639.

McCarthy, J. (1960), "Recursive Functions of Symbolic Expressions and Their Computation by Machine", CACM, vol. 3, no. 4, 184-195.

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

Tennent, R.D. (1977), "Language Methods Based on Semantics 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, 685-696.

Wirth, N. (1971), "Program Development by Stepwise Refinement", CACM, vol. 14, no. 4, 221-227.

*UNIX is a Trademark of Bell Laboratories.