Pascal — The Language and its Implementation
Edited by D. W. Barron
© 1981, John Wiley & Sons, Ltd.

# 2

# *Ambiguities and Insecurities in Pascal*

*J. Welsh, W. J. Sneeringer, and C. A. R. Hoare*

## 1. Introduction

On rare occasions in programming-language development there appears a programming language which is widely recognized as superior, and which propagates itself among discerning implementors and users solely by its merits, and without any political or commercial backing. ALGOL 60 [1] was such a language; Pascal [2] is another.

One characteristic of such superior languages is that they rapidly give rise to a host of suggested extensions, improvements, and imitations. From ALGOL 60 came ALGOL D [3], ALGOL W [4], ALGOL 68 [5], PL/I [6], SIMULA 67 [7], and Pascal itself. Pascal has been followed by the critique by Habermann [8], Concurrent Pascal [9], Pasqual [10], Modula [11], and Euclid [12]. It is one of the symptoms of the superiority of these languages that their original design remains superior to many of their successors, and even the authors themselves can find little to improve in formulating a revised version [13, 14, 15]. Thus the very superiority of the language may inhibit for a while the further progress of the art of language design.

One reason for this is that there is no immediate recognition of exactly what constitute the merits of the language. Indeed, the merits of ALGOL 60 have only recently been appreciated under the new name of *structured programming*. Similarly, most criticism of the language is rather superficial, concentrating on critics' favourite 'features' and 'facilities' which have been left out.

If future language designs, and indeed future users, are to benefit fully from the significant advances made by Pascal, it is essential that its defects, as well as its virtues, should be carefully identified and catalogued. The detailed, almost 'pettifogging' nature of the criticisms in this paper may be taken as a testimony to a belief that Pascal is at the present time the best language in the public domain for purposes of systems programming and software implementation. Nevertheless, these criticisms may lead to a better understanding of the definitional problems created in Pascal and to a better treatment of these problems in the languages which must inevitably follow it.

No consideration is given to changes in Pascal other than those necessary to overcome the ambiguities and insecurities identified.

Throughout this paper, the abbreviations *User Manual* and *Report* are used to stand for the first and second parts, respectively, of the *Pascal–User Manual and Report* [15]. There are several earlier versions of the *Report* [14]. The abbreviation *Axiomatic Definition* is also used for the formal definition of Pascal's semantics given by Hoare and Wirth [16].

## 2. Ambiguities

Ambiguities and omissions in the *Report* or *User Manual* are not mentioned in what follows if they can be easily resolved. The objective is to criticize the language, not the *Report* or the *User Manual*; but insofar as the language's features (or apparent intentions) create problems of definition, the *Report* must be considered as well.

### 2.1 Equivalence of types

Section 9.1.1 of the *Report* states that the two sides of an assignment statement 'must be of identical type', with certain exceptions involving reals and subranges. The phrase *identical type* is not defined and its meaning is not obvious. Much of Habermann's criticism of Pascal hinged on the omission from the *Report* of similar exception rules for other contexts in which subrange or real variables might or might not appear. As the *Axiomatic Definition* shows, the subrange problem can be resolved by the systematic introduction of implicit subrange-range transfers as context requires. However, the notion of type equivalence creates problems for other Pascal types too.

In the declarations,

```
type T =  array[1..10] of integer;
var A,B:  array[1..10] of integer;
     C:   array[1..10] of integer;
     D:   T;
     E:   T;
```

consider the following two possible definitions of equivalence of types;

*Name equivalence.* Two variables are considered to be of the same type only if they are declared together (as *A* and *B*) or if they are declared using the same type identifier (as *D* and *E*). Any type specification other than a type identifier creates a new type which is not equivalent to any other type. Thus $A$, $C$, and $D$ all have different types. Notice that primitive types are specified using type identifiers, so two variables will have the same type if they are both declared *integer*. This is called name equivalence because two variables that are not declared together can have the same type only if they are declared using the same type name.

*Structural equivalence.* Two variables are considered to be of the same type whenever they have components of the same type structured in the same way. Using this definition, all of the variables in the example above have the same type.

Name equivalence is quite a nuisance to the programmer, since he or she must often make up extra type names. On the other hand, name equivalence provides extra protection against type errors. Furthermore, structural equivalence causes a logical problem. Consider this example using structural equivalence:

```
var K: (male,female);
    L: (male,female);
```

Clearly the types of $K$ and $L$ are equivalent, since they have the same structure. However,

```
var M: (male,female);
    N: (female,male);
```

is illegal because the identifiers *male* and *female* are not unique. Distinguishing between these two cases will be difficult for the compiler. It cannot simply consider the construct (*male,female*) to be a declaration of the identifiers *male* and *female* as it could in the case of name equivalence. If it did, it would reject the legal declaration of identifier $L$.

Even worse, suppose that the type (*male,female*) is created and then the identifier *male* is used for an unrelated purpose in an inner block. Then, in a block inside both of these, the construct (*male,female*) is used again. Is it a reference to the first type? A new type? An error? There seems to be no good answer.

The use of structural equivalence also creates a problem with record types, which is illustrated by the following:

```
var F: record T,U: real end;
    G: record V,W: real end;
```

Do $F$ and $G$ have the same type? Either answer seems reasonable and consistent.

Structural equivalence creates a further dilemma for the implementor in relation to the **packed** prefix for structured types. Section 6.2 of the *Report* states that the prefix 'has no effect on the meaning of the program but is a hint to the compiler that storage should be economised even at the price of some loss in efficiency of access'. Presumably, therefore, a packed type is equivalent to an otherwise structurally equivalent unpacked one, and the compiler must permit, and generate code for, assignment or, worse still, actual-formal parameter correspondence between them. This problem does not arise with name equivalence since the syntax of ⟨*type*⟩ excludes the form **packed** ⟨*type identifier*⟩.

Name equivalence is not, however, without its problems. It precludes, for example, the assignment of a string constant to a variable of a corresponding string type. Section 4 of the *Report* states that a string constant of *n* characters has an implicit type

**packed array** [1..*n*] **of** *char*

With name equivalence this implied type cannot be equivalent to any other, so the string may only appear in certain limited contexts such as calls on the built-in procedure *write*. A similar problem arises with constructed sets, whose type is also implicitly specified; this problem is considered further in a later section of this paper.

Name equivalence also creates a potential confusion for the user of the type definition

**type** *T*1 = *T*2;

where *T*2 is the name of a type defined elsewhere. With name equivalence this will not produce a convenient local synonym for type *T*2 as might be expected, but a new type *T*1 which is not equivalent to type *T*2 in any context.

Clearly the current features of Pascal do not permit a simple choice between name or structural equivalence as defined. Some alternative or compromise equivalence definition must be adopted. In practice, of course, each implementation of Pascal has already made some choice. The ETH compiler (the compiler described by the *User Manual*) uses structural equivalence in most cases [17]. However, a scalar type declaration such as (*male*,*female*) is taken to be a declaration of the identifiers *male* and *female*, and therefore causes a message about a duplicate declaration if repeated in the same block. (It will create a new type equivalent to the first if used in an inner block.) Record types are equivalent if the corresponding field types are the same, but packed structured types are not equivalent to corresponding unpacked ones.

It is unsatisfactory that implementors should be left to make such decisions, since any diverenge in their choice imperils the portability of Pascal programs. The authors of Euclid, Pascal's most recent derivative, were clearly conscious of Pascal's deficiencies in this area. Although the definition of Euclid has been modelled on the Pascal *Report*, it incorporates an explicit definition of type equivalence based on the repeated replacement of type identifiers by the sequence of symbols appearing in their definition. Two types are equivalent if, in the sequences of symbols which they produce,

(a) corresponding occurrences of free constant identifiers (i.e. those not declared by these types) denote the same value, and
(b) corresponding symbols are otherwise identical.

The resultant definition of type equivalence is close to the structural equivalence suggested above. Whether this particular definition is the best for the language remains an open question, but the provision of *some* such explicit definition is an important requirement, both for Pascal and for any language which imitates its repertoire of data types.

### 2.2 Scope rules and one-pass compilation

One of the design objectives stated for Pascal was to enable efficient compilation of its programs. Although the *Report* does not say so explicitly, the language features appear to favour one-pass compilation as a means to this end, and implementors have assumed this to be the designer's intent. However, this implicit one-pass compilation capability creates some traps for the unwary, into which implementors have duly fallen.

In general, one-pass compilation requires that the declaration of an identifier precede all other references to that identifier. The Pascal *Report* does not specify at any point that an identifier's declaration must precede its use. It does, however, impose a rigid order on the different classes of declaration which are made within a block, thus:

```
⟨block⟩ ::= ⟨label declaration part⟩
            ⟨constant definition part⟩
            ⟨type definition part⟩
            ⟨variable declaration part⟩
            ⟨procdeure and function declatation part⟩
            ⟨statement part⟩
```

This has the effect of ensuring that constant identifiers are defined before they can be used in type definitions, that type identifiers are defined before they can be used in variable declarations, and that variable identifiers are declared before they can be used in the statement part as non-locals in nested procedures and functions. However, it does not guarantee declaration before use *within* the type definition or procedure declaration parts. For example, the following program segment is unacceptable to a one-pass compiler because the use of the identifier *complex* to declare type *matrix* precedes the declaration of *complex*:

```
type matrix  = array [1..10,1..10] of complex;
     complex = record realpart,imagpart: real end;
var  m: matrix;
     .
     .
     .
write(m[2,2].realpart);
```

Now consider the same program segment and assume that this declaration is in the containing block:

```
type complex = record re,im: real end;
```

There are at least two possible interpretations of this program by a one-pass system:

(1) The elements of *m* each have two real components with names *re* and *im*, since the outer declaration of *complex* was current when the declaration of *matrix* was scanned.
(2) The program is in error because the inner declaration of *complex* is one that should apply, and it follows the declaration of *matrix*.

This program is incorrect in either case. Under interpretation (1), the call to *write* is incorrect because the *realpart* is not the valid field name for the variable *m*. We are not just haggling over which statement receives the error message, however. If the field name *realpart* in the call to *write* were replaced by *re*, the resulting program would be correct according to interpretation (1) and incorrect according to (2).

Notice that (1) is the easier interpretation to implement. Each reference to an identifier is simply bound to tbe most recent declaration of that identifier. One way to implement (2) might be to bind the element type of *matrix* to the outer definition of *complex*, but record this binding so that an error can be declared when the inner definition of *complex* is scanned. The recorded binding has to be applied not only in the current scope but also in any enclosing scopes between it and the outer definition.

Unfortunately, interpretation (1) has some problems involving pointer types and (2) is the better interpretation. Consider the following examples:

```
type flight =
       record
         number: 0..999;
         firstpas: passenger;
         . . .
       end;
```

```
passenger =
   record
      flightbooked: flight;
      nextpas: passenger
      . . .
   end;
```

Each of the two types in the example refers to the other, so whichever type is declared second in the program will have its identifier referenced before it is declared. This is a case where the rule that identifiers must be declared before they are used is too restrictive to be practical, and Pascal implementations make an exception to accommodate this case. Pointer declarations, such as *passenger,* are allowed to precede the declaration of the identifier used. Fortunately, the compiler can allocate storage for a pointer without konwing what type of thing it will reference, since the size of a pointer does not depend on what it points at.

The problem with interpretation (1) is illustrated by the example above if there happens to be a type *passenger* declared in an outer block. In that case, interpretation (1) demands that the name *passenger* in field *firstpas* be bound to the outer definition of type *passenger.* This is very bad, because the meaning of a valid block can be changed by declaring an identifier in an outer block.

In fact the *Report* does exclude interpretation (1) since section 4 states that the 'association (of identifiers) must be unique within their scope of validity, i.e. within the procedure of function in which they are declared'. (No explicit definition of scope for main program identifiers is given.) However, the significance of section 4 is clearly not apparent to its implementors, since the ETH compiler itself follows interpretation (1), even to the extent of binding the pointer type *passenger* to a non-local instance of *passenger* if one exists.

A similar difficulty arises with mutually recursive procedures and functions. To retain one-pass compilation with checking of parameters the ETH compiler requires a **forward** declaration, which is not described in the *Report* or included in the syntax diagrams or BNF. It is described only in section 11.C of the *User Manual,* from which we take the following example:

```
procedure q(x:t); forward;
procedure p(y:t);
   begin
    q(a)
   end;
procedure q; (*parameters not repeated*)
   begin
    p(b)
   end;
```

```
begin
   p(a);
   q(b)
end.
```

The line

```
procedure q(x:t); forward
```

which must precede the procedure *p*, provides enough information so that the call to *q* from within *p* can be compiled.

These problems arise because the *Report* does not define any precise rules for the relative positions of the declaration and use of identifiers. Implementors of one-pass compilers must impose additional restrictions on the language definition or create the unsatisfactory implementation effects outlined above. If one-pass compilation is to be a language objective it should be made explicit in the language definition and an explicit declaration-before-use rule should be adopted, with whatever exceptions the language features may require.

With such a rule the rigid order which Pascal imposes on the constant, type, variable, and procedure declaration parts could be relaxed, allowing natural groupings of the types, variables, and procedures which manipulate them. The inability to make such groupings in structuring large programs is one of Pascal's most frustrating limitations.

Given such groupings, or modules, of course the additional controls on their mutual interaction such as those which Modula and Euclid provide are clearly desirable. The control over the use of identifiers which these languages offer is a clear indication that the current needs of programming have moved well beyond the implicit scope rules of simple block structure—and Pascal.

## 2.3 Set constructors

As was indicated earlier, the implicit types of string constants and constructed sets create problems in defining type equivalence. However, the implicit type definition creates additional problems in the representation of sets.

According to section 8 of the *Report*, 'Expressions which are members of a set must all be of the same type, which is the base type of the set.' This leads to the conclusion that the type of the set constructor [1,5,10..19,23] is **set of** *integer*, since section 4 of the *Report* makes it clear that the types of 1, 5, 10, 19, and 23 are *integer*. However, section 14 of the *Report* states that: 'The implementer may set a limit to the size of a base type over which a set can be defined. (Consequently, a bit pattern representation may reasonably be used for sets.)' Since the apparent base type in the example is *integer* and the size of type *integer* is larger than any reasonable limit, one might conclude that the example is illegal in at least some implementations. The example comes from section 8 of the *Report*, so it seems fair to say that the *Report* is confusing, if not ambiguous.

In practice the conflict is yet another which is resolved by an implicit range to subrange transfer. Given that a limit on the size of base types exists, the compiler may assume that the intended base type of (1,5,10..19,23] is some subrange of the integers, and apply an implicit range to subrange transfer to its member values. The problem is that the intended subrange is not apparent, which in turn has consequences for the representation of the set.

Using a bit pattern representation for sets, the type

**set of** 20..29

is represented by a bit pattern very much like the type

**packed array** [20..29] **of** *boolean*

where the element *n* of the array is *true* if and only if *n* is in the set. The trouble is that the base type of [1,5,10..19,23] has not been specified, so the compiler does not know what the bounds of its Boolean array should be.

Implementations overcome this problem by imposing an additional limit on the base types of sets. For example, in the ETH compiler the limit of the size of a base type is 59, so the compiler knows that the Boolean array can be no larger than 59. However, the compiler also needs its upper and lower bounds. The ETH compiler therefore adds the restriction that each element of any set of integers must be between 0 and 58. This rule allows any set of integers to be represented by an array with bounds 0 and 58. A similar rule applies to sets of non-integers. In that case, no element *e* is allowed unless *ord(e)* <=58. This solution has the consequence that apparently representable set types such as

*dates* = **set of** 1939..1945

are excluded by current implementations.

For implementations which choose, or are forced by short word lengths or byte orientation, to use multilength representations of sets, the implicit type fo the set constructor presents an additional problem. Either all sets over subranges of a given type must use the same length of representation or the required length of a constructed set must be deduced from context. The extreme case occurs when the empty set [] (which has no implicit base type at all) occurs as an actual parameter of a formal procedure or function (which provides no contextual indication of the representation required).

All these problems can be avoided by requiring an explicit specification of the base type of every set constructor. For example, given a set type *digits* = **set of** 0..9

the constructor notation used might be

*digits*(1,3,5)

This makes the programmer write a bit more, but allows the base type of a set to be any scalar type that does not have too many elements. Not only are the restrictions simpler and less constraining, but the language is cleaner because every set constructor has a type which can be determined during compilation without any use of context. A version of Pascal using such a constructor, and a multiword representation of sets, has been implemented [18] and shown to provide a more flexible, and more efficient, set-manipulation facility. A similar notation has now been adopted in Euclid.

This notation also reconciles the set constructor with the name equivalence convention for types discussed earlier. A similar solution for string constants might be considered. Given a string type

*message* = **packed array** [1..16] **of** *char*

a constant of the type might be written thus

*message* (*'illegal operands'*)

In this case the additional burden on the programmer may be unacceptable in contexts where named type specification is unnecessary, e.g. in calls to the built-in procedure *write*, and some default for omitting the type name and parentheses may be appropriate.

## 3. Insecurities

For the purposes of this discussion, an insecurity is a feature that cannot be implemented without either (a) a risk that violations of the language rules will go undetected or (b) runtime checking that is comparable in cost to the operation being performed.

Pascal has fewer insecurities than most comparable languages. For example, it is not possible to use a pointer to access a dynamic variable of the wrong type. This error is caught during compilation because each pointer can only point at variable or a single type. The remarkable thing about Pascal is that the number of insecurities is small enough to make it worthwhile to prepare a list in the hope that future research will lead to languages with even fewer or perhaps no insecurities.

## 3.1 Variant records

Pascal allows variant records with and without tag fields. An example of a variant record with a tag field is

```
v: record  area:real;
      case  s:shape of
            triangle:(side:real;
                      inclination,angle1,angle2:real);
            circle:  (diameter:real)
      end
```

The field *area* always exists, but whether *diameter* exists or not depends on whether the value of the tag field *s* is *circle* or not. A version of this record without a tag field can be created by omitting '*s*:' after the symbol *case*.

If there is no tag field, then the variant record is inevitably insecure. Either *angle*1 or *angle*2 could be referenced when it is not present and there is no way to catch the error, even at run time. This is bad, because such an error is likely to be difficult to find.

The compiler could insert a tag field even when the programmer does not request it, but it would be misleading and pointless to allow the programmer to omit the tag field if the compiler included it anyway. The introduction to Pascal of variant records without tag fields must be regarded as a retrograde step, to be regretted by Pascal users and avoided by the designers of future languages.

Given that a tag field is present in all variant records, a runtime check is still required to achieve security. However, the runtime check can be avoided when code like the following is used to reference the variant part:

```
case  v.s of
      triangle: begin (references to v.side, etc.) end;
      circle:   begin (references to v.siameter) end
end
```

since the value of the tag field when the references occur is known when the program is compiled. This is done in SIMULA 67 with an *inspect when* statement, which is similar to the *case* statement above. A similar modification to Pascal has been investigated [19], which showed that direct violations from within the *case* construct were easily detected, but that detection of indirect changes of the variant by reassignment of the entire record variable, possibly during procedure calls from within the case, was impractical. The *Report* (section 9.2.4) does outlaw such changes. with a **with** statement but this restriction is equally impractical to enforce by compiletime or runtime checking.

Euclid incorporates an explicit construct which enables direct variant violations to be detected at compile time. For more general reasons of program verification Euclid's definition also goes to considerable lengths to enable variable overlaps, such as might cause an implicit change of variant, to be detected. Whether the added complexity of the rules required and the added restrictions which they impose on the programmer are an acceptable price to pay for variable access security, may be shown by experience of implementing and using Euclid. The rules and restrictions involved cannot be readily added to the current framework of Pascal.

## 3.2 Functions and procedures as parameters

When a Pascal formal parameter is a function or a procedure, the language does not require or even permit the programmer to specify the number and types of any parameters. The following example, which is due to Lecarme and Desjardins [20], illustrates a program which contains an error that cannot reasonably be detected at compile time:

```
procedure p (procedure q);
   begin q(2,'a') end;
procedure r(x :boolean);
   begin write(x) end;
begin p(r) end.
```

The apparent solution is to allow full specification of parameters in this case, as is done in ALGOL 68. The normal syntax for parameter specification is excessive for this purpose since it includes specification of names of the formal parameters, and these names are not required. Lecarme and Desjardins proposed a syntax for specifying the types without giving names; With their syntax, the first line of the example above is written

```
procedure p (procedure q(integer,char));
   begin q(2,'a') end;
```

which gives enough information for the error to be detected during compilation.

To enable the correct parameter passing code to be generated for their calls, Pascal currently allows procedures and functions passed as parameters to take value parameters only. Given an adequate notation for expressing the parameter requirements of formal procedures or functions, this restriction can in principle be relaxed. The notation required is more complicated than that of Lecarme and Desjardins, however, since it must distinguish between variable and value parameters. If procedure parameters which themselves take procedure parameters are allowed, the notation must also provide a nested, and potentially recursive, specification of parameter requirements.

The Pascal compiler for UNIVAC 1100 computers, developed at DIKU in Copenhagen, incorporates an extension which meets these requirements [21]. Formal parameter lists can be defined and named in a separate parameter declaration part of each block. The parameter requirements of actual procedures may then be specified by reference to a named parameter list, and those of formal procedures must be specified in this way. In the DIKU system formal parameter names are always included in the parameter list specification, so that procedures sharing a parameter specification must use the same formal parameter names as well.

## 3.3 Range violations

As in most compiled languages, accessing an element of a Pascal array is insecure if an index is out of bounds. Since all languages have this problem, it is appropriate to try to solve it with hardware. The extra hardware cost is quite small. The descriptor mechanism of the ICL 2900 series computers provides an implicit bound check during array access but, while it works well for the arrays allowed in languages such as FORTRAN or ALGOL 60, it is inadequate for some of the array structures permitted in Pascal [22].

Array access is just one of a number of contexts in which a value outside a permitted range can arise in Pascal. Others are assignment to a subrange variable, case selection, set membership creation and testing, and indeed overflow in integer and real arithmetic. While it is unreasonable to hope to exclude by language design the possibility of all such violations, designers must aim to reduce the cost of their runtime detection. It should be noted that Pascal's provision of enumerated and subrange types is a significant step in this direction. Each use of a variable of an enumerated type removes a potential insecurity by ensuring that the finite set of values which the variable may take is verified at compile time. For a subrange variable runtime verification of the values taken may be necessary but, assuming thses checks are made, other more frequent and hence more expensive checks may be avoided at each point where the variable value is used. A Pascal compiler which exploits this technique has been constructed [23] for ICL 1900 computers, and has shown that for simple array manipulations runtime subscript checking can be eliminated, or reduced to insignificance.

## 3.4 Uninitialized variables

Uninitialized variables are also difficult to detect, and all hardware detection mechanisms known to us are quite expensive. Possible solutions are to require that every variable be initialized when it is declared or that every variable be assigned in such a way that the compiler can easily verify that there are no references to uninitialized variables. The latter might work very well in a language without jumps, and deserves further investigation.

### 3.5 Dangling references

Accessing of dynamic variables (those found via pointers) is not secure because the storage for the dynamic variable may have been released. This is a very common insecurity for which Pascal allows no obvious solution.

It can be argued that Pascal's pointer is a low-level facility provided for use in those situations for which the high-level data constructs are inadequate and that it is unreasonable to expect security from a low-level facility. Whatever the philosophical validity of this argument it is little consolation to a programmer whose pointers go wrong!

Euclid offers an optional security against such errors by enabling reference counts to be maintained for collections of dynamically allocated variables. Storage release is then an implicit operation occuring when a reference count reaches zero, rather than an explicit programmable action. Maintaining reference counts is, of course, a considerable overhead if applied to every pointer variable assignment. The success of the Euclid proposal depends on the degree to which the compiler can detect those program segments which use local pointer variables to trace a dynamically allocated structure without altering the non-local reference pattern in any way. Reference counting code can then be avoided for the pointer manipulation within the segment.

## 4. Conclusion

At the time that Pascal was first designed and developed, the most fashionable languages in the learned and practical world were ALGOL 68 and PL/I. The discovery that the advantages of a high-level language could be combined with high efficiency in such a simple and elegant manner as in Pascal was a revelation that deserves the title of breakthrough. Because of the very success of Pascal, which greatly exceeded the expectations of its author, the standards by which we judge such languages have also risen. It is grossly unfair to judge an engineering project by standards which have been proved attainable only by the success of the project itself, but in the interests of progress, such criticism must be made.

Of the criticisms made in this paper, some identify shortcomings of Pascal which can readily be made good by minor changes to the language or its definition. Others indicate problems for which there is no easy solution within the current framework. As a language which attempts to overcome most of the problems listed, Euclid deserves a special mention, though it should also be pointed out that no implementation of Euclid has yet been reported. Unfortunately, Euclid achieves its goals at the expense of a significant loss of simplicity and elegance in the language definition. Whether this trade-off is inevitable or whether some future breakthrough can restore elegance and simplicity without loss of security is a question which language designers must ponder for some time to come.

**Acknowledgement**

The research on this paper was supported in part by a grant from the Science Research Council of Great Britain.

## References

1. Naur, P. (Ed.), Report on the algorithmic language ALGOL 60, *Communications ACM*, **3**, 299–314, 1960.
2. Wirth, N., The programming language Pascal, *Acta Informatica*, **1**, 35–63, 1971.
3. Galler, B.A., and Perlis, A. J., A proposal for definitions in ALGOL, *Communications ACM*, **10**, 204–219, 1967.
4. Wirth, N., and Hoare, C. A. R., A contribution to the development of ALGOL, *Communications ACM*, **9**, 413–432, 1966.
5. van Winkngaarden, A. (Ed.), Report of the algorithmic language ALGOL 68, *Numerische Mathematick*, **14**, 79–218, 1969.
6. IBM, *PL/I(F) Language Reference Manual*, Order No. C28–8201, IBM, 1969.
7. Birtwistle, G., *et al.*, *Simula Begin*, Auerbach, 1975.
8. Habermann, A. N., Critical comments on the programming language Pascal, *Acta Informatica*, **3**, 47–57, 1973.
9. Brinch Hansen, P., The programming language concurrent Pascal, *IEEE transactions on Software Engineering*, **1**, no. 2, 1975.
10. Tennent, R. D., *Pasqual: A Proposed Generalisation of Pascal*, Technical Report No. 75-32, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1975.
11. Wirth, N., Modula: A language for modular multiprogramming, *Software–Practice and Experience*, **7**, 3–35, 1977.
12. Lampson, B. W., *et al.*, Report on the programming language Euclid, *ACM Sigplan Notices*, **12**, no. 2, 1977.
13. Naur, P., Revised report on the algorithmic language ALGOL 60, *Communications ACM*, **6**, no. 1, 1963.
14. Wirth, N., *The Programming Language Pascal* (revised report), Berichte der Fachgruppe Computer-Wissenschaften Nr.5, ETH, Zurich, 1973.
15. Jenson, K., and Wirth, N., *Pascal–User Manual and Report*, Lecture Notes in Computer Science, no. 18, Springer-Verlag, Berlin, 1974.
16. Hoare, C. A. R., and Wirth, N., An axiomatic definition of the programming language Pascal, *Acta Informatica*, **2**, 335–355, 1973.
17. Ammann, U., The Zurich implementation, chap. 7 in this volume.
18. Copeland, C. J., *Extensions to Pascal*, M.Sc. dissertation, Queen's University, Belfast, 1975.
19. Sinte, P. W. C., *Recursive Data Structures in Pascal*, M.Sc. dissertation, Queen's University, Belfast, 1975.
20. Lecarme, O., and Desjardins, P., More comments on the programming language Pascal, *Acta Informatica*, **4**, 231–243, 1975.
21. Steensgaard-Madsen, J., *Procedures as Monitors in Sequential Programming*, DIKU, Copenhagen, Denmark.
22. Rees, M., *et al.*, Pascal on an advanced architecture, chap. 13 in this volume.
23. Welsh, J., Two 1900 compilers, chap. 10 in this volume.