A comparison of PASCAL and ALGOL 68

A. S. Tanenbaum

Vakgroep Informatica, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, The Netherlands

PASCAL and ALGOL 68 language features are critically examined, compared and contrasted with each other. The emphasis is on the differences between the two languages, primarily in the areas of data types and statement types. Shortcomings in both languages are pointed out.

(Received June 1977)

1. Introduction

Although ALGOL 60 was never widely used outside academia, it did at least bequeath to the world some interesting progeny: PASCAL and ALGOL 68. In a sense both might claim to be its rightful successor. It is therefore interesting to compare and contrast PASCAL and ALGOL 68, since they have a common ancestor, and were designed with similar goals in mind: to encourage good programming practices, to communicate algorithms among people, to be efficiently executable on a variety of existing computers, and to be useful for teaching students programming. Such a comparison is the purpose of this paper. Since the terminologies used in their respective reports to describe ALGOL 68 and PASCAL are about as compatible as an ILLIAC IV and an Intel 8080, we will attempt to disappoint the opposing pedants equally.

The reader is assumed to have some familiarity with both languages; the references at the end of this paper provide more detail for those who are not. When referring to language features, we are referring to the language described by the defining reports (Jensen and Wirth, 1974, Van Wijngaarden et al., 1975) and not to any specific implementation. It should be kept in mind that not all compilers correctly and fully implement the language they claim to implement. Since certain language design issues impinge on all implementations of that language, we will occasionally discuss the relation between language design and implementation, and try to indicate how implementers have usually dealt with a certain issue, where there is a consensus.

An example of the distinction between language definition and implementation is the question of garbage collection. It is widely said that ALGOL 68 has garbage collection and PASCAL has not. In fact, neither defining documents mentions garbage collection (an implementation issue) at all. Nevertheless in ALGOL 68 circles, lack of it is considered a serious restriction, whereas PASCAL users do not generally expect it. Thus the issue is more one of tradition, difficulty of implementation and folklore than of a specific difference in the official language definition.

2. Lexical conventions and syntax

Another 'gray area' is the way boldface words such as begin, if, case and while are represented in machine readable form. ALGOL 68 specifically allows all these boldface words as variable identifiers; PASCAL specifically forbids them as identifiers. As a result, a full ALGOL 68 implementation must provide two distinct type founits, such as upper and lower case, or have a stopping convention ('begin', "begin",...begin) for boldface. One founit is sufficient for PASCAL programs. It should be noted that user defined modes (types) in ALGOL 68 are written in boldface in ALGOL 68 publications and in lightface in PASCAL publications, viz. 'dog' in:

ALGOL 68:  mode dog = struct (bool barks, bytes)
PASCAL: type dog = record barks, bytes:boolean end

This means that there are potentially an infinite number of boldface words in ALGOL 68, which makes reserving all of them in advance impossible. PASCAL's reserved word list is finite, so the report can (and does) simply prohibit their use as identifiers.

A closely related issue is the use of space 'inside' identifiers. In ALGOL 68 they are allowed but ignored; in PASCAL they are prohibited. Consider the following program fragment in a context where BEGINK, NEND, N, and K are all integer variables, and upper case is the only available founit:

BEGIN K := N END

Unfortunately the line is ambiguous, either being an assignment of NEND to BEGINK, or a short block. To avoid such ambiguities, PASCAL implementations are forced to prohibit spaces inside identifiers. With ALGOL 68 and stropping there is no confusion between:

"BEGIN"' K := N"END"' (a block)

and

BEGIN K := N END  (an assignment to BEGINK)

Superficially, the languages have similar syntaxes and statement types, with differences that are often a matter of taste e.g.

ALGOL 68: proc, bool, int, char letter PASCAL: procedure, boolean, integer, letter: char

ALGOL 68 has explicit delimiters (if,...fi, do,...od, etc.) whereas PASCAL uses begin end for grouping, except for repeat and case.

Both languages support block structure and nested procedures. One difference is that in PASCAL variables can only be declared in procedure headers (including the main program), whereas in ALGOL 68 they can be declared in any begin end block, within the condition of an if statement, in a subscript, and in a lot of other strange places.

One major difference between the languages is that ALGOL 68 has an expression language and PASCAL is not. Every construction, except declarations, but including those that look like statements, can in principle yield a value in ALGOL 68, and be used accordingly. Thus

begin int i,j; read ((i,j)); if i < j then i else j + 1 fi end

can be used in ALGOL 68 anywhere an integer is allowed (e.g. in a subscript). PASCAL makes a clear distinction between statements and expressions; they are not interchangeable.

ALGOL 68 has four different symbols for delimiting comments: #, & co and comment. Since the start-of-comment symbol is the same as the end-of-comment symbol, if the programmer inadvertently forgets one of them, program text will be treated as commentary and commented as program text. Unless the compiler being used is very clever, this leads to pages and pages of useless error messages. In PASCAL the opening and closing comment delimiters are different, { and }, which mitigates this problem substantially.

Both languages allow manifest constants (giving a constant...
a symbolic name), e.g.

PASCAL: \texttt{const smallpi} = 3.13
ALGOL 68: \texttt{real smallpi} = 3.13

However, ALGOL 68’s facility (identity-declarations) is much more general, allowing arbitrary expressions of any data type on the right hand side. PASCAL allows only scalar and string constants.

3. Primitive data types
In this section we will take a look at the standard data types that can be regarded as built in; in the next one we will look at how both languages allow users to construct their own.

3.1 Data types more or less identical in both languages
There are 4 primitive data types that have close counterparts in the other language:

PASCAL: \begin{tabular}{llll}
integer & real & boolean & char
\end{tabular}
ALGOL 68: \begin{tabular}{llll}
int & real & bool & char
\end{tabular}

3.2 Naked machine words
The designers of both languages realised that many programmers are often smitten with an irresistible, overwhelming urge to pack as much information as possible into a single machine word. Accordingly, both languages provide facilities through which the compiler writer can appease this demand, if he so desires. ALGOL 68 has primitive data types \texttt{bits} and \texttt{bytes}, the former similar to a boolean array, and the latter similar to a character array. The number of elements in these arrays is determined by the compiler writer, not the user or the language definition. It is expected that most implementers will choose to make \texttt{bits} and \texttt{bytes} objects fit neatly into a machine word. Operations are provided to get at the individual \texttt{bits} and \texttt{characters}, respectively.

PASCAL has a more general concept: the \texttt{packed} array. A compiler writer is free to treat \texttt{packed} arrays as being identical to unpacked arrays, since there are no semantics associated with packing. On the other hand, he also has a golden opportunity to store a \texttt{packed} boolean array of the appropriate length as a single machine word. Similarly a \texttt{packed} character array of the suitable length could also be stored in one machine word. The PASCAL concept makes it possible for portable programs since a 36 element boolean array can certainly be implemented on a 32 bit word machine, but an ALGOL 68 program that assumes an object of type \texttt{bits} has 36 elements will not work when moved to a machine where bits have only 32 elements.

3.3 Primitive data types unique to PASCAL
PASCAL has two basic types with no ALGOL 68 counterparts: subrange and scalar types. There is no way for the ALGOL 68 programmer to declare:

\begin{Verbatim}
type fourbitint = 0 . . 15;
type = (tooth, head, stomach, back)
\end{Verbatim}

In the former case, the ALGOL 68 programmer has little choice but to accept one of the (implementation dependent) lengths, such as \texttt{short short int}. In the latter case the closest he can come is:

\begin{Verbatim}
mode ache = short short int; \& or some other length
ache tooth = 1, head = 2, stomach = 3, back = 4;
\end{Verbatim}

In both cases the list can be stepped through symbolically, although in ALGOL 68 it is the programmer’s responsibility to make sure that numbers are stepped without errors to identifiers:

\begin{Verbatim}
for pain := tooth to back do calldoctor od
\end{Verbatim}

Many PASCAL implementations provide the option of having subrange variables checked for validity after each assignment; ALGOL 68 does not provide similar facilities.

3.4 Primitive data types unique to ALGOL 68
ALGOL 68 has 4 types absent from PASCAL, which look primitive to the uninitiated, but about which the sophisticated user knows better: \texttt{compl}, \texttt{string}, \texttt{sema} and \texttt{format}. That complex numbers and variable length character strings are not present in PASCAL, is probably attributable to the user’s ability to easily create the former, and the compiler writer’s difficulty in implementing the latter efficiently. Semaphores and formats are not needed, since PASCAL lacks parallel processing and formatted I/O.

3.5 Multiple precision arithmetic
Although PASCAL does not really support multiple precision arithmetic, its philosophy suggests a radically different approach to that of ALGOL 68. A minor semantic extension to PASCAL would allow subranges to be arbitrarily large, viz.

\begin{Verbatim}
type doubleint = \ldots
\end{Verbatim}

In other words, the user tells the compiler what he wants, and the compiler determines how much precision to use. ALGOL 68, in contrast, offers the user \texttt{int}, \texttt{long int}, \texttt{long long int}, \texttt{short int}, etc. In other words, the compiler writer determines what choices will be on the menu, and the user picks the one he prefers. Using the environment enquiries, it is (in principle) possible for an ALGOL 68 program to determine which length of \texttt{int} to use and then to proceed accordingly, but making the determination can be tricky, since overflows must be avoided and even if successful, procedures and operators must be provided to handle all the possible lengths. In short, the PASCAL approach is better for writing portable programs. However, ALGOL 68 provides for various lengths of reals (and complex numbers) which PASCAL does not address.

4. Methods for constructing new data types
Both languages support user defined data types. Below we examine some of the similarities and differences.

4.1 Arrays
PASCAL and ALGOL 68 both allow single and multi-dimensional arrays of arbitrary type elements (including arrays as elements). In PASCAL the bounds are both fixed and part of the type. Thus a procedure that accepts a 10 element integer vector as parameter will balk at a 9 element integer vector. There is no way to write a PASCAL procedure that accepts an arbitrary sized integer vector as parameter, much to the chagrin of PASCAL users. In ALGOL 68 formal bounds are neither fixed nor part of the type; a procedure that takes an integer vector as parameter will accept one of any size.

ALGOL 68 also has flexible arrays, variables whose size dynamically adapts to the size of the object assigned to them. PASCAL does not have flexible arrays. In practice, flexible arrays are of little value except for character strings. The most obvious use for flexible arrays would be to be able to extend an existing array. This, however, is not easy.

ALGOL 68 permits arbitrary cross sections of arrays to be manipulated as though they were entire arrays, e.g. if \( X \) is a 1 by 10 matrix, \( X[2:5, 3:7] \) can be assigned to, passed as a parameter, etc. PASCAL does not allow arbitrary cross sections to be handled as entities, although rows of two dimensional arrays, and certain planes of higher order arrays may be manipulated as units. Columns and subrows of two dimensional arrays cannot be handled as units, however.

PASCAL allows scalar types as subscripts as well as integers. In ALGOL 68 subscripts can only be integer expressions.

4.2 Records
Records (structures) in both languages consist of a sequence of
named fields, each of which may have an arbitrary type. The syntax for accessing a field is different, however:

PASCAL:  \texttt{dirt}.\texttt{lump}  
ALGOL 68:  \texttt{lump of dirt}

4.3 Deviant types

PASCAL allows records certain of whose fields have several alternatives. These are called record variants. A similar effect can be achieved in ALGOL 68 using unions. An important difference, however, is that in PASCAL the programmer is responsible for keeping track of which variant is currently in use (e.g., by using the field \texttt{h}, below). If he makes a mistake, the results are unpredictable. This is one of the very few areas of PASCAL which is "unsafe", i.e., can lead to programs making wild stores all over the place, etc. The ALGOL 68 union is completely safe; the syntax has been designed so that all improper accesses can be caught at compile time. Consider the following PASCAL program:

```
program berserk (output);
type trouble = record
  case b: boolean of
    true: (i:integer);
    false: (pt:*real);
  end;
var x:trouble; k:integer;
begin for k := 1 to 1000 do
  begin
    x.b := true;
    x.i := 7*k;
    {so far so good}
    x.b := false;
    {unnecessary}
    x.pt := k
    {wild store}
  end;
end.
```

All of the statements in the above program are individually legal, but during execution it will randomly store garbage around memory and quite possibly destroy itself in the process.

In ALGOL 68 the program would be written as follows:

```
begin
  mode trouble = union(int, ref real):
  trouble x;
  for k from 1 to 1000 do
    x.i := 7*k;
    case x in
      (int i): skip,
      (ref real pt): pt := k {ok if pointer \texttt{pt} defined here}
      esac
  od;
end;
```

In the PASCAL program we assign integer values to the variant and then access them as if they were pointers. This usage is similar to equvaluating an integer and a real in FORTRAN, storing a real in the variable, and then fetching it as an integer, causing the bit pattern of the real to be interpreted as an integer. Although in general PASCAL provides excellent security against the program corrupting itself, in this situation it falls flat on its face. Run time checking is made exceedingly difficult by the fact that a record variant field can be passed as a parameter to a procedure and assigned to. The procedure doing the assignment has no way of knowing that it is changing a field of a variant record.

In ALGOL 68 this kind of aliasing is impossible. The only way to get at the contents of \texttt{x} above is via the conformity-clause, which ensures that the programmer cannot store data of one type in the union, and access it as another. In the above program the \texttt{skip} statement will be executed 1000 times.

4.4 Pointers

Bad pointers can give rise to such obscure errors that both languages have taken care to bind pointers to specific data types. If \texttt{m} is an arbitrary data type, then PASCAL's \texttt{p : \textit{type}} and ALGOL 68's \texttt{ref m p} both declare a pointer \texttt{p} that can only point to objects of type \texttt{m}. The PASCAL-ALGOL 68 method of strong typing of pointers can be contrasted with PL/I's method, in which any pointer can point to any location in memory, and often does, much to the dismay of the programmer.

ALGOL 68 has automatic dereferencing; PASCAL does not.

Example:

```
PASCAL: var p : \textit{integer}; i : \textit{integer}; \ldots; i := p;
ALGOL 68: ref int p; int i; \ldots; i := p
```

In the assignment, an explicit dereferencing of the pointer is needed in PASCAL, but not in ALGOL 68. However, explicit dereferencing is possible in ALGOL 68, using a cast, e.g., \texttt{i := int(p)}. In PASCAL the user must specify how many dereferencings are needed, and where; in ALGOL 68 the user may specify with a cast what the required type is, and the compiler figures out how to get there (i.e., the coercion path).

An important difference between the languages is the class of objects that may be pointed at. An ALGOL 68 pointer can point at any object of the correct type. A PASCAL pointer, in contrast, can only point at a heap object (an object generated by "\textit{new}") and cannot point into the stack, at a local object. Furthermore, PASCAL pointers can only point at the start of a heap object, and not at a field in the middle, as an ALGOL 68 pointer can. These differences have important consequences for security and efficiency.

Consider the following ALGOL 68 program

```
begin
  ref int pt;
  begin
    int k;
    pt := k {illegal assignment \texttt{\textit{pt}} defined here}
    \texttt{\textit{pt}} undefined here
  end;
end;
```

ALGOL 68 requires that in an assignment to a pointer, the scope of the object being pointed to be at least as old of that of the pointer itself. In other words, the object being pointed at must not suddenly vanish while the pointer still exists. In the above program, the assignment is therefore illegal.

The catch is: it is not possible to detect all illegal assignments at compile time. Imagine the assignment above being replaced by the procedure call: \texttt{assign(pt, k)}, which actually did the assigning. Consequently either some assignments will have to be checked for scope validity at run time, or the programmer will have to get used to living in fear of wild stores.

PASCAL almost eliminates the problem by the simple expedient of requiring all pointers to point only to heap objects. Since heap objects have the whole program as scope, they can never mysteriously vanish into thin air, as above. The only loophole is pointing to an object whose storage has already been reclaimed by a call to \texttt{dispose}. Of course, only allowing pointers to point to the start of heap objects may inconvenience the programmer, or force him to use heap objects where the ALGOL 68 programmer could use local ones. In many implementations heap objects require more overhead than local objects, although this is not always so.

Both languages have a special symbol, \texttt{nil}, which can be assigned to pointers for terminating lists. In ALGOL 68, however, the mode of \texttt{nil} is peculiar and its use is highly counterintuitive, usually forcing the programmer to resort to a subterfuge to disambiguate the mode. The existence of \texttt{nil} also implies that wild stores can occur in both languages.
unless nil checking is performed at run time. Some compilers use an illegal address for nil to make the hardware do the checking free.

4.5 Procedures
ALGOL 68 procedures are objects; they can serve as array elements, fields of structures, be pointed at, be returned by functions, etc. PASCAL procedures are not objects; they can be called and passed as parameters, but that is all. Actually, proc in ALGOL 68 is a structuring method that combines the types of the parameters and results into a new type, e.g. a procedure taking an integer parameter and delivering a real result is an object of type proc (int) real. Since procedures are not data objects in PASCAL, they do not have types.

PASCAL makes a clear distinction between procedures that return explicit values (functions) and those that do not (procedures). In ALGOL 68 all procedures return something, void at the very least, so there is no qualitative difference between functions and procedures. This is a consequence of ALGOL 68 being expression oriented and PASCAL being statement oriented.

ALGOL 68 procedures can return any type, a direct consequence of its orthogonal design. In contrast, PASCAL functions can only return a restricted collection of types, which does not include arrays or records.

4.6 Files
ALGOL 68 files are structures with exactly 20 fields, the most important of which is the book, which specifies a fictional three dimensional character array (technically flex [ ] flex [ ] flex [ ] char). It consists of a certain number of pages, each with a certain number of lines, each with a certain number of characters. Procedures are provided to convert numbers to character strings into the current position in the book. Random access can be achieved on (some) files by setting the current (page, line, character) values to a specified value. The other fields of a file keep track of its format, current position, string terminator, procedures for exception handling, and so forth.

The PASCAL model of a file is completely different. A file is not regarded as an object, as in ALGOL 68, but as a strange sort of one dimensional array. There are character files, integer files, real files, etc. Associated with each file is a buffer variable, which contains the next element of the file. The program can read or write this variable. The standard procedures get and put read the next element into the buffer variable, and write the buffer variable onto the end of the file respectively.

Both languages have a concept of ‘current position’ within an open file. In ALGOL 68 it is explicit, in the form of (page, line, character); in PASCAL it is implicit, since the only operations are read and write the next element. By setting (page, line, character) the ALGOL 68 programmer can randomly access characters anywhere in a file; PASCAL does not have random access files.

In addition to character files, ALGOL 68 also has binary files, on to which data can be written and later reread. Pointer values cannot be written, even on binary files. PASCAL also provides for binary output by allowing the element of a file to be any value, even a pointer, or a record containing pointers. This is also a potential breach of security, like the record variant problem, and for that reason is not implemented by many compilers.

It is interesting to note that, in PASCAL, a procedure is a primitive concept, and a file is a structuring mechanism like in array or record. In ALGOL 68 just the reverse is true: a procedure is a structuring concept and a file is an object.

1.7 Sets
PASCAL allows the user to create sets of any scalar type.

The maximum number of elements in a set is not specified in the report but rather is implementation dependent. This has unfortunate effects on program portability. It would have been better to have allowed the user to specify how large he wished his sets to be, with the compiler choosing the appropriate number of machine words in which to represent a set. Operations are provided for computing set union, difference, intersection, and membership. ALGOL 68 has no language construct similar to the set, but similar operations can be performed on objects of mode bits. However, the ALGOL 68 programmer must keep track of the meaning of the various bits explicitly. In PASCAL one just manipulates the elements, and the compiler worries about storage allocation.

4.8 Type declarations
ALGOL 68 mode declarations and PASCAL type declarations achieve the same effect: giving a name to a user created data type. Since all array bounds in PASCAL are constants, all objects of a given type are of the same size, except for records with variant fields. In ALGOL 68, symbolic bounds in mode declarations are allowed and evaluated whenever the name of the mode is encountered, so objects of the same mode may have different sizes.

The ALGOL 68 revised report precisely describes under what circumstances two modes are equivalent and may be used interchangeably. For example:

mode m1 = union (int, real);
mode m2 = union (real, int)

are equivalent. Objects of mode m1 may be assigned to variables of mode m2, and vice versa.

On the other hand, the PASCAL definition is silent about the question of type equivalence. Whether

type node = record
  val : integer;
  left : node;
  right : node
end;

type noad = record
  val : integer;
  left : noad;
  right : noad
end;

represent the same type is left unanswered. May a programmer declare a formal parameter to be of type node, and pass as actual parameter a noad variable? Each implementer is forced to decide for himself what to do, with dire consequences for program portability. It is perhaps worth mentioning that type equivalence is not so serious for programs written by a single person, but when a large number of people are involved in a project, minor discrepancies like this can easily creep in.

5. Scope rules
Block structured languages must have carefully defined scope rules to determine how long a value continues to exist after its creation. In nearly all block structured languages, including PASCAL and ALGOL 68, variables local to a procedure (or block) come into existence when the procedure is entered and cease to exist when the procedure is exited. These local
variables are generally implemented using a stack discipline. ALGOL 68 local variables may be initialised in their declarations; PASCAL local variables may not be.

In ALGOL 68 it is possible to declare variables local to any block. These variables come into existence when the block is entered, and cease to exist when the block is terminated. PASCAL does not allow variable declarations except at the procedure level.

For a number of applications, some form of dynamic storage allocation is needed in which the objects generated continue to exist even after the procedure in which they were created has been exited. These objects are said to be on the heap, rather than on the stack. Their scope is the entire program. In PASCAL heap objects are created using the built-in function new, as in new(p), which generates a new object and assigns its address to p. (The type of the new object is determined by p; if p has type ?m, then an object of type m is created.)

If p has variant fields, their tags may be specified, presumably to allow the appropriate amount of storage to be allocated. However if the storage allocator actually uses this information to allocate less than maximum storage, subsequent changes to the variant field can cause the program to destroy itself, viz:

```plaintext
program overwrite(output);
type vector = array [1..10000] of integer;
type f = record
case integer of
1: (small: integer);
2: (large: vector);
end;
end;

var x: t; ptr: ?t; biggie: vector;
begin
new(ptr, 1);
ptr. large := biggie
end.
```

Of course if the programmer subsequently intends to reassign the variant field, as above, he should not specify a tag in the call to new, but the above program is nevertheless legal according to the defining report. It is hard to imagine an implementation which both performs the intended optimisation and works correctly.

In ALGOL 68 heaps objects are created by heap generators, e.g. p := heap [1..n] real. In addition to heap generators, ALGOL 68 has explicit local generators. Objects created dynamically using local generators, vanish when the current procedure is exited. This means that the stack discipline (which is often more efficient than the heap of discipline) can be used for them. PASCAL does not have local generators.

Neither language requires nor forbids garbage collection, but most ALGOL 68 implementations have it, and most PASCAL implementations lack it. PASCAL does provide a standard procedure, dispose, which allows the programmer to indicate that a certain heap object is no longer needed. It is expected that the storage management system will use this information to help recover unused storage. Part of the reason why most PASCAL implementations lack full garbage collection is the insecurity of record variants, as discussed in Section 4.3. In general, a run time garbage collector could not tell which variant was "current" by looking at the tag field, since it might well be incorrect.

One possible solution (widely used in ALGOL 68 compilers for unions) is to have the compiler create and update a 'hidden' field which at all times indicates which variant is current. However, this is easier said than done. A variant field of a record can be passed to a procedure and assigned to it. To update the 'hidden' variable the procedure will have to know which of its parameters are variant fields. Although this information can be passed along with the parameter, some overhead is introduced in doing so. Worse yet, if a record has several nested variants, keeping track of exactly which variant is in which state can be excruciatingly complicated.

The possibility of writing pointers onto a file, and then reading them back also makes garbage collection difficult: what happens if garbage collection occurs between the time the pointer is written out and read back? The pointer will be invalid when read back.

6. Expressions

Both languages allow expressions consisting of variables and constants separated by the usual arithmetic and relational operators. However, ALGOL 68 allows many kinds of expressions not possible in PASCAL. Some examples of valid ALGOL 68 expressions follow:

```plaintext
if i < j then i else j fi
case i in 4, j + 2, k esac
begin int i; read(i); i end
```

Each of the above can be used in any context an integer is expected, e.g. in a subscript.

Furthermore, since ALGOL 68 provides a mechanism for for users to define prefix (monadic) and infix (dyadic) operators, all kinds of strange expressions can result. A user could define the + operator on 2 files (for example to concatenate them) and then use the expression file2+file3. In principle a dyadic operator can accept two objects of arbitrary types as operands and yield a third object of yet another type as result. In PASCAL, only real, integer, boolean, scalar and set operators and results are possible. There are no built in operators on arrays, records, pointers or files (except for (in)equality operators on pointers and unpacked arrays, and relational operators on packed character arrays). Furthermore there is no way to create any, either. The ability to create one's own operators, facilitates programming with abstract data types. In this style of programming, data structures are not manipulated directly by the program, but only by using certain operators defined along with the data structures. If the operator definitions are correct, the integrity of the data structures is always assured, no matter what errors the rest of the program contains, providing, of course, that the data structures are only manipulated by the 'official' operators.

Another kind of expression present in ALGOL 68 and absent in PASCAL is the row-display, an explicit list of the elements of an array. The ALGOL 68 programmer alone can write:

```plaintext
x[1..5] := (9, 8, 7, 6, 0)
```

Likewise, the ALGOL 68 programmer can use a structure-display to assign simultaneously to all the fields of a structure.

Another difference between ALGOL 68 and PASCAL is that in the former the precedence of all infix operators, both built in and user defined, are under user control. If the user wants p and q or r to mean p and (q or r) instead of (p and q) or r, that is easily arranged by simply defining the precedence of or to be higher than that of and. In PASCAL the operator precedences are immutable.

Mixed mode arithmetic is allowed in both languages; however, neither allows real values to be assigned to integer variables. An explicit conversion is required; in ALGOL 68 one can use the standard operators round and enter; in PASCAL one can use the standard functions round and trunc.

7. Statements

7.1 Assignment statements

Assignment works essentially the same way in both languages, including assignment of entire arrays or records in one statement (i.e. a := b, where a and b are both arrays). One difference, however, is that ALGOL 68 allows multiple assignments such
as $i := j := k := 0$, whereas PASCAL does not. Of course, since the types in the two languages do not correspond, the assignments do not correspond either, e.g. there are no assignments to procedure variables in PASCAL, or to set variables in ALGOL 68.

7.2 if statements
The if statements differ in both syntactic and semantic ways. ALGOL 68 has an explicit closing delimiter, fi, which eliminates the need for begin end blocks to enclose statement groups in either the then part or else part. In other words, the then part does not consist of one statement (including a compound statement) as in PASCAL, but rather all statements up to a matching else, fi or elif. In addition to making programs more readable by eliminating unnecessary begin end blocks, the dangling else problem is solved in PASCAL.

if . . . then if . . . then $S_1$ else $S_2$

would have been ambiguous had the report not simply defined the problem away by fiat.

The semantic difference alluded to comes from the ALGOL 68 programmer's ability to declare variables in the if clause and then use them in the then and else parts. The value of this facility is debatable: some programmers believe in declaring all variables as locally as possible and others argue that declaring all variables in a procedure together improves 'findability' hence readability.

7.3 Case statements
Multiway branches in the form of case statements are present in both languages, but in different forms. The index of the ALGOL 68 version must be an integer expression with a non-negative value. The index, say $i$, selects the $i$-th clause for execution. The order in which the clauses are written is crucial. In PASCAL the index can be any scalar expression. Each clause is labelled by one or more values. The index is evaluated and the clause labelled by its value is executed. The order of the clauses is immaterial. In ALGOL 68 an optional out clause may be specified, which is executed if the index is out of range. The effect of an out of range index in PASCAL is not defined; in ALGOL 68 the whole statement is skipped, unless there is an out clause. PASCAL has no out clause.

7.4 Looping statements
ALGOL 68 combines the for and while statements into a single statement, whereas they are distinct in PASCAL. The various pieces of the ALGOL 68 version are all optional, except the do, i.e. in

for $i$ from $1$ by $1$ to infinity while true do . . . od

the for, from, by, to and while clauses are individually optional, with the defaults given above, except that if the for part is omitted there is no controlled variable. The PASCAL statements are simpler, but all the parts are required.

In PASCAL there is no by part as such, the only possibilities being $+1$ or $-1$. If $-1$ is intended, to is replaced by downto is in

for $i := 10$ downto $1$ do . . .

by telling the compiler in which direction the counting is to proceed, a more efficient implementation is possible, as well as a more easily understood program. In the ALGOL 68 construction

for $i$ from $n_1$ by $n_2$ to $n_3$ do . . . od

the compiler cannot tell at compile time in which direction the counting is to proceed, so it must insert run time code to find out. On the other hand, the ALGOL 68 programmer has more lexicability, since the direction can be determined dynamically, based on the input data.

The controlled variable in ALGOL 68 is implicitly declared (as an integer) by its use in the for statement. Assignments to it are illegal, so it cannot be modified by the programmer. Its scope is the loop, making it inaccessible outside the loop. If a variable with the same name is declared outside the loop, that does not affect the controlled variable, its value or scope in any way. The controlled variable in PASCAL must be either an integer or some other scalar type. In contrast to ALGOL 68, where its declaration is impossible, the PASCAL controlled variable must be declared (e.g. in the procedure in which it appears). In contrast to ALGOL 68, it is accessible after the loop is finished, but its value is undefined.

In ALGOL 68 the expressions following by and to are evaluated before the loop execution begins; subsequent changes to their component variables are allowed but have no effect on the termination condition. In PASCAL the programmer is forbidden from causing the to part to change its value during loop execution. Programs in which the to part is changed are incorrect, but many implementations will fail to give an error message.

PASCAL has a repeat statement, which tests at the end of the loop instead of at the start. The ALGOL 68 while statement can achieve the same effect, albeit in a somewhat awkward way:

while $S_1$; $S_2$; . . . $S_n$; condition do skip od

Neither language provides for any loop escape mechanism such as the exit constructions in BLISS (Wulf et al., 1971). ALGOL 68 has something called a completer, but it is not really good for much.

7.5 Procedure calls
Procedures can be called in both languages by writing the procedure name, optionally followed by a parenthesised parameter list. Despite the superficial similarity, there are a number of major differences between ALGOL 68 and PASCAL procedure calls. To start with, objects of any type are allowed both as parameters and results (of functions) in ALGOL 68. In PASCAL there are several restrictions on parameters:

1. files passed by value
2. elements of packed arrays and records as var parameters
3. parametric functions and procedures with other than value parameters

Further arrays, records, files and sets are prohibited as function results. This is a clear example of ALGOL 68's orthogonality and PASCAL's lack thereof. In ALGOL 68 the choice of which procedure to call can be made dynamically:

if phase = 1 then read else print fi ((x, y, z))

In PASCAL the procedure to be called cannot be computed. Another area of difference is the parameter mechanism. ALGOL 68 has a single parameter mechanism that is essentially a strict call by value. The 'value' of an object of type ref $m$ is the address of some object of type $m$. In this way a procedure may return results via its parameters, even though all parameters are called by value. Formal parameters are definitely not initialised local variables, as in PASCAL and attempts to assign to any parameter whose type is not ref something are illegal and will be flagged by the compiler.

PASCAL, on the other hand, has three distinct parameter mechanisms: value, reference, and procedure (including function). Value parameters are evaluated before the call and are used to initialise the corresponding formal parameters, which thereafter may be modified by the called procedure. Values cannot be returned to the caller using them. Reference (var) parameters are simply variable names (including elements of arrays and fields of records), which use call-by-reference.
Procedure and function names are considered as a special case.

One idiosyncracy of PASCAL is that although the type of all value and reference formal parameters must be declared in full, function and procedure parameters are for some peculiar reason exempt from this general principle. Rather than making the implementation easier or more efficient (which might be a legitimate reason to deviate from a general rule), the lack of full specification for procedure and function parameters makes the implementation more difficult and less efficient! Worse yet, a programmer can inadvertently call a procedure parameter with the wrong number of parameters, which in some implementations will result in a wild jump followed within microseconds by a program crash. ALGOL 68 does not have this problem. Passing procedures as parameters is perfectly safe.

Within a PASCAL function, values are assigned to the function by assigning to the function identifier. In ALGOL 68 the value of a function is simply the value of the last unit executed.

7.6 go to statements

Both languages have go to statements. In both cases the language designers realised that their control structures were insufficient and that some kind of mechanism was needed to abort the normal statement flow when an error was detected. PASCAL requires all labels to be explicitly declared in a label declaration which has some nuisance value at least and aids 1 pass compilation. PASCAL labels are numeric, and ALGOL 68 labels are non-numeric.

7.7 Miscellaneous statements

The PASCAL with statement has no ALGOL 68 counterpart. It does not add any expressive power to the language but it saves a little writing occasionally and makes efficient code generation easier in some cases. It also introduces some ambiguity since a with statement may specify several variables, and each of the records broken open may have some identical fields:

```
  television: record colour, portable: boolean end;
  program: record debugged, portable: boolean end;
  with television, program do portable := true;
```

In PASCAL, declaration must precede application in order to make one pass compilation possible. ALGOL 68 allows application to precede declaration; one pass compilation of the full, unmodified language is therefore impossible. If procedure A calls procedure B, and B calls A, the first one to appear in the source text cannot be compiled until the mode of the second one is known, which implies at least two passes. Many PASCAL compilers have a forward declaration to get around the problem, but that is, strictly speaking, not proper PASCAL.

8. Input/Output

Input/Output is performed in both languages by a set of built in procedures. In both cases these procedures have certain special properties not available to user defined procedures. The PASCAL and ALGOL 68 I/O procedures are so different that it is difficult to compare them feature by feature; it is like comparing BASIC to PL/I. Printed output in PASCAL is achieved by the procedures put, which writes a buffer variable onto a file, or write and writeln, each of which write an expression, or a list of expressions and strings onto a file. The field width and number of places after the decimal point may be specified where appropriate. That is pretty much all there is to say about PASCAL printed output.

ALGOL 68 is another story. The procedures print and put output a list of expressions and strings, as in PASCAL, but without user control over fields widths. The procedure printf, on the other hand, provides an extremely general and powerful formatted output facility including user control over character conversion, page handling, data errors, end of file conditions, etc. The definition of formatted I/O occupies 33 full pages in the ALGOL 68 report! (Whether this is an advantage or a disadvantage will be left for the reader to decide.)

Input in PASCAL is accomplished by the procedures get, read, and readln (and eoln and eof to check for end of line and end of file respectively). Formatted input is impossible, which means that if the user wants complete freedom to read arbitrary input data, he must read it one character at a time, with all character-to-number conversions performed explicitly by the user. ALGOL 68 again has both the simple read procedure and the very elaborate readf for formatted input.

9. Language description

ALGOL 68 is defined by a massive, often totally opaque, but exceedingly precise defining document. PASCAL is defined by an informal report, rather in the style of the ALGOL 60 report. They also use different kinds of grammars: ALGOL 68 uses a two level, context sensitive grammar, known as a vW grammar; PASCAL uses a context free, BNF grammar. As a consequence there are an infinite number of terminal productions of (program) in PASCAL which are, in fact, syntactically incorrect, such as

```
  program wrong (output):
  var i: integer;
  begin i := 3-14; i := true end.
```

The analogous construct in ALGOL 68 is not a terminal production of (program). In other words, the mere fact that a Program obeys the PASCAL grammar does not mean that it is syntactically correct. It must also meet a number of conditions imposed or hinted at or implied by the English text, with all the ambiguity that entails. Needless to say, ambiguity is not a highly desirable attribute for a programming language definition to have. On the other side of the coin, a completely unambiguous definition of the syntax, such as ALGOL 68 has, is of little value if the users of the language do not understand a word of it.

In both defining documents the semantics are described by English text, in PASCAL informally, relying on the reader's experience with computers and programming languages in general, and in ALGOL 68 in a stylised, often cryptic form. As with the syntax, PASCAL's definition is much easier to read and frequently sloppy or ambiguous whereas ALGOL 68's is rarely ambiguous, assuming you can figure out what it is trying to say.

In one area ALGOL 68 is more precise and at least as readable as PASCAL: the definitions of the standard operators, procedures and environment. The ALGOL 68 revised report contains a 'standard-prelude' in which nearly all of the built in operators, procedures, modes and environment enquiries are defined in ALGOL 68 itself. It is through the standard prelude mechanism that the language proper is kept surprisingly small. Modes such as string, compl, sema and format are defined there in ALGOL 68, so the reader can see exactly what they are and how the operators on them are defined. The PASCAL document has nothing comparable to the standard prelude. For example, the PASCAL report does not specify whether round (−1.5) is −1 or −2. Some implementers will choose −1 and other implementers will choose −2, with unfortunate consequences for program portability.

10. Summary and critique

ALGOL 68 is clearly a larger scale undertaking than PASCAL in every dimension: it has more features, is applicable to a wider class of problems, requires more study to learn well,
and certainly demands considerably more effort to produce a good compiler for it. Whether the additional power offered is worth the additional effort needed to be able to use it, depends on the user and the problem.

In addition to many common features, both languages suffer from a number of the same shortcomings. We now briefly recapitulate some of these, not to demean the significant accomplishments of the PASCAL and ALGOL 68 designers, but to point to some 'unfinished business' for subsequent language designers to deal with.

The simplest problem to remedy is the inadequacy of the control structures, to which the presence of the goto statement attests. What is needed is a general mechanism for abnormal loop termination. Many suggestions have been offered in the literature (Wulf, Russell and Habermann, 1971; Knuth, 1974), including one by PASCAL's creator (Wirth, 1977).

Lack of orthogonality is another shortcoming, one from which PASCAL suffers more than ALGOL 68. In PASCAL the restrictions on actual parameters and function results are particularly obvious. In ALGOL 68 the rules about where primaries, secondaries, tertiaries and units may be used, and the rules specifying which coercions are allowed in which context are anything but orthogonal.

Both languages have features which make it possible for programs to run amok in unpredictable and difficult to track down ways. In PASCAL, security can be breached by accessing a record variant by a field which is not 'current', and by reading in pointers. In ALGOL 68 violations of the scope rules can lead to trouble. Use of a nil pointer is also a security problem. Appropriate run time checks can prevent programs from running wild, but these cost both time and memory. A language design which could prevent them at compile time would be much preferable. But at least in ALGOL 68 programs can be kept from running amok by simple run time checks, the myriad problems with the record variant construction in PASCAL make such checks difficult.

The visibility and protection rules for identifiers in both languages are too liberal. All the variables declared in a procedure are automatically accessible to any procedure declared inside it. There is no way to prevent a variable from being inadvertently modified by an inner procedure which has no business using it. A better default rule would be no access unless a variable is explicitly declared as visible. The PASCAL/ALGOL 68 visibility rules make it difficult to declare an abstract data type consisting of a data structure and operators on it in such a way as to prevent access to the data structure except by using the operators.

A related problem is the difficulty in passing read only data to a procedure. For a simple object it can be passed by value, but if the data structure is a tree or graph this is inconvenient. Such data structures typically consist of a collection of nodes, each node holding a value and some pointers. To give access to a procedure, a pointer to one node is passed to the procedure. Even if the pointer is passed by value, there is no way to limit the damage the procedure can do to the data structure. Many operating systems allow a user to pass a file to another user for the purposes of reading only, but not writing. A similar (or better) protection system is lacking in both PASCAL and ALGOL 68.

The last point we wish to discuss is machine independence. Both languages make serious attempts to achieve this, but both fall somewhat short of the mark. PASCAL's restriction on sets to a cardinality which is machine dependent is a glaring example. The bits and bytes modes of ALGOL 68 and the crude way multiple precision is dealt with are equally bad. A reasonable approach would be to allow the programmer to specify what he needs (viz. PASCAL subranges). If this happened to mesh well with the machine's word length, it would be implemented efficiently. If it did not mesh, the compiler would use two or more machine words for the data type. In this way all programs would run on all machines, albeit more efficiently on some than on others.

11. Acknowledgements
I would like to thank Jack Alanen, Robert Fraley, Dick Grune, Ed Keizer, Kees Koster, O. LeCarme, Bruce Leverett, C. H. Lindsey, Alan Mackworth, Rudolf van Bottenburg and Reind van de Riet for their critical comments.

References